DEA *Systèmes Informatiques Répartis*
Tronc Commun *Conception par Objets*
et Prototypage d'Applications Concurrentes

Objets, Composants, Agents

Copie transparents en :
`http://www-poleia.lip6.fr/~briot/cours/composants-agents-sir01-02.pdf`

Jean-Pierre Briot

Thème OASIS
(Objets et Agents pour Systèmes d'Information et Simulation)
Laboratoire d'Informatique de Paris 6
Université Paris 6 - CNRS
Jean-Pierre.Briot@lip6.fr

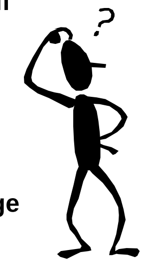
Plan général

- I - Objets pour la Programmation Concurrente et Répartie
  - Enjeux, problèmes, approches
  - Approche applicative
  - Approche intégrée
  - Approche réflexive
- II - Composants
  - Des objets aux composants
  - Architectures Logicielles
  - Frameworks et design patterns
- III - Agents
  - Motivations
  - Des objets aux agents
  - Différents types d'agents
  - Principes et techniques

I - Objets pour la Programmation Concurrente et Répartie

Inflation des termes, et des concepts ??

objet **tâche** **réflexion** **acteur** **méta-programmation** **objet actif** **futur** **processus** **objet distribué** **transmission asynchrone** **moniteur** **anomalie d'héritage** **agent** **transaction** **CORBA** **Java** **réplication** **atomicité** **temps virtuel**



Objectif

- Rappeler en quoi les concepts d'objet (de facto standard actuel de la programmation "classique, i.e. séquentielle et centralisée) offrent une bonne fondation pour la programmation parallèle et répartie
- Analyser et classifier les différents types d'articulation entre :
  - programmation par objets
  - programmation parallèle et répartie
- Nous considérons trois approches principales :
  - applicative *(structuration sous forme de bibliothèques)*
  - intégrée *(identification et unification des concepts et mécanismes)*
  - réflexive *(association de méta-bibliothèques de mise en œuvre à un programme - idée : réifier le contexte du calcul, de manière à pouvoir adapter un programme à différents environnements et contraintes de calcul)*
- Analyser
  - les limites d'une transposition naïve des concepts d'objet, ou plutôt des techniques d'implantation, à la programmation parallèle et répartie
  - de possibles solutions

Objets pour la Programmation Parallèle et Répartie

- Exposé fondé sur une étude menée en collaboration avec Rachid Guerraoui, EPFL, Suisse
- Articles de référence :
  - «Objets pour la programmation parallèle et répartie», Jean-Pierre Briot et Rachid Guerraoui,
    - » Technique et Science Informatiques (TSI),15(6):765-800, Hermès, France, juin 1996.
    - » dans «Langages et modèles à objets», édité par Amedeo Napoli et Jérôme Euzenat, Collection Didactique, INRIA, 1998.
  - «Concurrency and distribution in object-oriented programming», Jean-Pierre Briot, Rachid Guerraoui et Klaus-Peter Löhr, ACM Computing Surveys, 30(3):291-329, septembre 1998.

(sous)-Plan

- Applications informatiques : enjeux actuels et futurs
- Concepts d'objet
  - Potentiel (concurrence et répartition) et limites
- Objets, parallélisme et répartition : 3 approches
- Approche applicative
  - Principes, Exemple, Bilan
- Approche intégrée
  - Dimensions d'intégration (objet actif, objet synchronisé, objet réparti)
  - Exemples, Limitations
- Approche réflexive
  - Principes, Exemples, Bilan
- Conclusion

Enjeux actuels et futurs

- De la programmation séquentielle, centralisée, en monde clos
  ... à la programmation parallèle, répartie, de systèmes ouverts
  - ex : travail coopératif assisté par ordinateurs (CSCW)
  - ex : simulation répartie multi-agent
- Décomposition fonctionnelle (logique) : Concurrence
  vs Mise en oeuvre (physique) : Parallélisme
  - intrinsèque (ex : multi-agent, atelier flexible)
  - a posteriori (temps de calcul)
- Répartition
  - intrinsèque (ex : CSCW, contrôle de procédé)
  - a posteriori (volume de données, résistance aux pannes)
- Système ouvert
  - reconfigurable dynamiquement, ex : Internet
  - adaptation à l'environnement, ex : contraintes de ressources (temps, espace..)

Concepts d’objet

- objet : module autonome (données + procédures)
- protocole de communication unifié : transmission de messages
- abstraction : classe (factorisation) d’objets similaires
- spécialisation : sous-classe (mécanisme d’héritage)

- encapsulation : séparation interface / implémentation
- gestion dynamique des ressources

- *concepts suffisamment forts* : structuration et modularité
- *concepts suffisamment mous* : généricité et granularité variable

Concurrence potentielle

- *Simula-67* [Birtswistle et al.’73]
  - *body* d’une classe : corps de programme exécuté lors de la création d’une instance
  - *coroutines* : suspension (`detach`) et relance (`resume`)

- Objets <-> Processus [Meyer, CACM’9/93]
  - variables
  - données persistantes
  - encapsulation
  - moyens de communication

- Contraintes technologiques et culturelles ont fait régresser ces potentialités parmi les successeurs directs de Simula-67

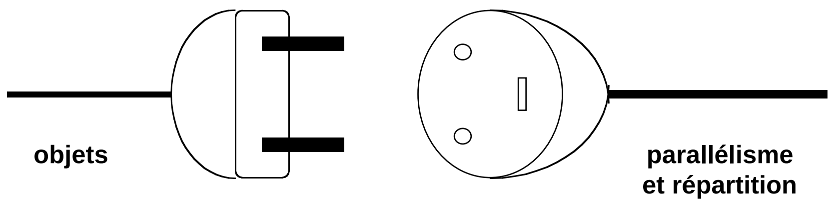
Répartition potentielle

- Objet = unité naturelle de répartition
  - unité fonctionnelle
  - transmission de messages
    - » indépendance services / implémentation (*encapsulation*)
    - » indépendance services / localisation (*transparence*)
  - autonomie et relative complétude facilite migration/duplication

- Architecture client/serveur <-> Objet
  - analogue
  - MAIS dichotomie client/serveur est *dynamique* chez les objets
    - » un objet envoie un message : *client*
    - » le même objet reçoit un message : *serveur*

Limites

- Mais malgré ses potentialités les concepts d’objet ne sont pas suffisants pour aborder les enjeux de la programmation parallèle et répartie :
  - contrôle de concurrence
  - répartition
  - résistance aux pannes



Travaux développés en parallèle

- Diverses communautés
  - programmation parallèle
  - programmation répartie
  - systèmes d'exploitation
  - bases de données

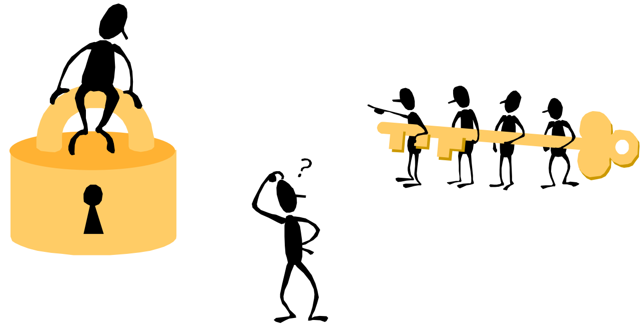
  ont développé différents types d'abstractions :
  - synchronisation
  - transactions
  - communication de groupes
  - ...

  pour aborder de tels besoins

Articulation

- La question est par conséquent :

Comment doit-on lier les concepts d'objet
aux acquis et enjeux de la programmation parallèle et répartie ?



Une classification des approches possibles

- Nous distinguons trois approches principales :
  - approche ***applicative***
    - » application *telle quelle* des concepts d'objet à la conception de programmes et systèmes parallèles et répartis
    - » processus, fichiers, ressources... sont des objets
    - » bibliothèques / frameworks
    - » Ex : *Smalltalk* [Goldberg et Robson'89], *Choices* [Campbell et al., CACM'9/93]
  - approche ***intégrée***
    - » identification et unification des concepts d'objet avec les concepts de la programmation parallèle et répartie
      - object = activité -> objet actif
      - transmission de message = synchronisation /et/ invocation distante
    - » ex : *Actors* [Agha'86], *Java RMI*
  - approche ***réflexive***
    - » séparation entre fonctionnalités (programme générique) et mise en œuvre (modèle d'exécution, protocoles de synchronisation, de répartition, de résistance aux fautes...)
    - » protocoles exprimés sous la forme de bibliothèques de méta-programmes/objets
    - » ex : *CLOS MOP* [Kiczales et al.'91], *OpenC++* [Chiba, OOPSLA'95], *CodA* [McAffer, ECOOP'95]

Complémentarité des approches

- Ces approches <u>ne sont pas en compétition</u>
- Elles ont des objectifs/niveaux complémentaires
  - approche applicative destinée aux <u>concepteurs de systèmes</u> :
    identification des abstractions fondamentales
    utilisationdes classes et de l'héritage pour structurer, classifier, spécialiser/réutiliser
  - approche intégrée destinée aux <u>concepteurs d'applications</u> :
    langage de haut niveau uniforme (minimum de concepts)
    -> maximum de transparence pour l'utilisateur
  - approche réflexive destinée aux <u>concepteurs de systèmes adaptables</u> :
    les concepteurs d'application peuvent spécialiser <u>dynamiquement</u> le système selon les besoins propres de leurs applications

Approche applicative

- Principes
  - appliquer *tels quels* les concepts d'objet à la structuration et la modularité de systèmes complexes
  - bibliothèques et frameworks
  - les différentes abstractions sont représentées par des classes (ex : en *Smalltalk*, processus, sémaphore, fichier...)
  - l'héritage permet de spécialiser <u>statiquement</u> un système générique
    (ex : dans *Choices*, sous classes concrètes correspondant à différents formats de fichiers, réseaux de communication, etc.)
  - les différents services sont représentés par différents objets/composants spécialisés (ex : systèmes d'exploitation à «micro-kernel», e.g. *Chorus* *[Rozier et al.'92])*
- Gains
  - compréhensibilité
  - extensibilité
  - efficacité

Smalltalk

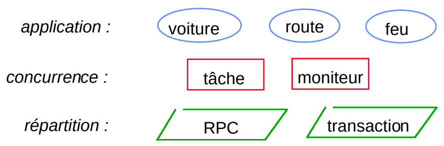
- Langage de programmation par objets minimal
- Riches bibliothèques de classes représentant :
  - constructions du langage (ex : structures de contrôle, `ifTrue:ifFalse:`)
  - ressources (messages, multi-tâche, compilateur...)
  - outils de l'environnement (ex : browser, debugger...)
- Concurrence
  - processus (tâches) (`Process`)
  - séquenceur (`ProcessorScheduler`)
  - sémaphores (`Semaphore`)
  - communication (`SharedQueue`)
  - *aisément extensibles ,* (ex :*Simtalk* [Bézivin, OOPSLA'97], *Actalk* [Briot, ECOOP'92])
- Répartition
  - communications (*sockets Unix*, *RPCTalk*...)
  - stockage (*BOSS*) -> persistance, encodage...
  - briques de base pour construire divers services répartis (*DistributedSmalltalk*, *GARF* [Mazouni et al., TOOLS'95] , *BAST* [Garbinato et al, ECOOP'96]...)

Autres exemples

- C++
  - bibliothèque de threads : C++, *ACE* [Schmid'95]
  - bibliothèque de répartition : *DC++* [Schill et Mock, DSE'93]
  - *Choices* [Campbell et al., CACM'9/93]
    - » classes abstraites : `ObjectProxy`, `MemoryObject`, `FileStream`, `ObjectStar`, `Disk`
    - » spécialisables pour des environnements spécifiques (fichiers Unix ou MS, disque SPARC, mémoire partagée...)
- Beta
  - bibliothèques de répartition [Brandt et Lehrman Madsen, OBDP-LNCS'94]
    - » Classes `NameServer`, `ErrorHandler`
- Eiffel
  - bibliothèques pour parallélisme de données (SPMD)
    - » structures de données abstraites répartissables en *EPEE* [Jezequel, JOOP'93]

Bilan

- Avantages : structuration, modularité, extensibilité
- Objectif de fond : dégager les abstractions minimales
  - synchronisation, atomicité, persistance, transaction...
- Limitations :
  - le programmeur a deux (ou même trois) tâches distinctes :
    - » programmer son application en termes d'objets
    - » gérer le parallélisme et la répartition
      - également par des objets,
      - mais ***PAS LES MÊMES !!!***



  - possible lourdeur
    - » ex : classe Concurrency [Karaorman/Bruno, CACM'9/93]
    - » encapsule activité ainsi que transmission de message distante et asynchrone
    - » ***MAIS*** impose une certaine dose de manipulation explicite des messages
  - (manque de) transparence
  - complexité (trop de dimensions différentes et indépendantes à gérer)

Approche intégrée

- Principes :
  – fusion des concepts d'objet avec les concepts de la programmation parallèle et répartie
  – offrir au programmeur un cadre conceptuel (objet) unique
  – plusieurs dimensions d'intégration possibles :
    » objet <-> activité -> <u>objet actif</u>
      • ex : *Acteurs*
    » activation <-> synchronisation -> <u>objet synchronisé</u>
      • transmission de messages : synchronisation appelant/appelé
      • au niveau de l'objet : synchronisation des invocations
      • ex : *Guide* [Balter et al., Computer Journal'94], *Arjuna* [Parrington et Shrivasta, ECOOP'88], *Java [Lea'97]*
    » objet <-> unité de répartition -> <u>objet réparti</u>
      • ex : *Emerald* [Jul et al.'98]
- Gains
  – simplicité

Approche intégrée (suite)

- Ces trois dimensions sont relativement indépendantes entre elles
- Ex : Java

objet actif	NON	un thread est un objet mais tout objet n'est pas un thread
objet synchronisé	OUI	à chaque objet un verrou (en fait un moniteur) est associé
objet réparti	NON	OUI avec Java RMI

Objet actif

- <u>objet = activité</u>
  – une activité : <u>sériel</u>
  – plusieurs activités
    » <u>quasi-concurrent</u> (ex : *ABCL/1* [Yonezawa'90])
    » <u>concurrent</u> (ex : *Actors* [Agha'86])
    » <u>ultra-concurrent</u> (ex : acteur non sérialisé)
- objet est <u>réactif</u> <-> activité (tâche/processus) est <u>autonome</u>
  – dans l'union, qui l'emporte ??
    » objet actif réactif (ex : *Actors*)
    » objet actif autonome (ex : *POOL* [America, OOCP'87], *CEiffel* [Löhr, OOPSLA'92])
- acceptation implicite ou explicite de messages
  – implicite (ex : *Actors*)
  – explicite
    » concept de <u>body</u> (hérité de *Simula*), ex : *POOL*, *Eiffel//* [Caromel, CACM/9/93]

Objet synchronisé

- synchronisation au niveau de la transmission de messages
  – transmission de messages : synchronisation implicite appelant/appelé (transmission *synchrone*)
  – transparent pour le client
  – dérivations/optimisations :
    » transmission *asynchrone*, ex : *Actors*
    » transmission avec réponse anticipée (*future*), ex : *ABCL/1*, *Eiffel//*
- synchronisation (des invocations) au niveau de l'objet
  – synchronisation intra-objet
    » en cas de concurrence intra-objet (multiples invocations)
    » ex : multiples lecteurs / un écrivain
  – synchronisation comportementale
    » dynamicité des services offerts
    » ex : buffer de taille bornée, le service `put:` devient temporairement indisponible pendant que le tampon est plein
    » transparent pour le client
  – synchronisation inter-objets
    » ex : transfert entre comptes bancaires, transaction

Formalismes de synchronisation

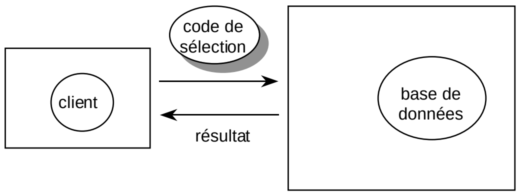
- Origines : systèmes d'exploitation, parallélisme, bases de données
- Intégration relativement aisée dans un modèle objet
  - formalismes centralisés, associés au niveau des classes
    - » *path expressions* (specif. abstraite des entrelacements possibles entre invocations)
      - ex : *Procol* [Lafra'91]
    - » *body* (procédure centralisée décrivant l'activité et les types de requêtes à accepter)
      - ex : *POOL*, *Eiffel//*
    - » *comportements abstraits* (synchronisation comportementale)
      - empty = {put:}, full = {get}, partial = empty U full
      - ex : *Act++* [Kafura, ECOOP'89] *Rosette* [Tomlinson, ECOOP'88]
  - formalismes décentralisés associés au niveau des méthodes
    - » *gardes* (conditions booléennes d'activation)
    - » compteurs de synchronisation
      - ex : *Guide*
    - » *Java* : verrou (lock) au niveau de l'objet avec mot clé `synchonized` au niveau des méthodes

Objet réparti

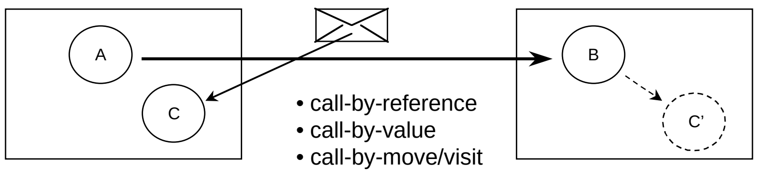
- objet : unité indépendante d'exécution
  - données, traitements, et même ressources (si objet actif)
  - transmission de messages conduit à la transparence de la localisation
  - autonomie et relative atomicité de l'objet facilite migration et duplication
- association de l'invocation distante à la transmission de messages
  - *Java RMI*
- association des transactions à la transmission de messages
  - synchronisation inter-objets et résistance aux pannes
    - » *Argus* [Liskov'83]
- mécanismes de migration
  - meilleure accessibilité, ex : *Emerald (call by move)*
- mécanismes de réplication
  - meilleure disponibilité (dupliquer les objets trop sollicités)
  - résistance aux pannes (ex : *Electra* [Maffeis'95])

Emerald

- Code mobile
  - rapprocher (code) traitement des données
  - ex : SQL



- Objet mobile
  - PostScript (code + données constantes)
  - Emerald [Black et al. IEEE TSE 87]

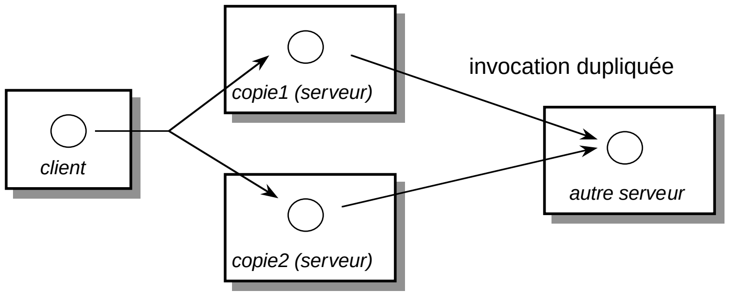


Limite 1 : Spécialisation de la synchronisation

- spécialisation des conditions de synchronisation
  - approche naturelle : utiliser l'héritage
  - *MAIS* cela ne marche pas si bien ! (<u>inheritance anomaly</u> [Matsuoka RDOBCP'93])
  - formalismes centralisés -> le plus souvent redéfinition complète
  - formalismes décentralisés -> peut induire des redéfinitions nécessaires
    - » ex : compteurs de synchronisation
      - nouvelle méthode en exclusion mutuelle -> clause à rajouter dans toutes les méthodes
      - solution Java : méthodes qualifiées `synchronized` en exclusion mutuelle entre elles
    - » ex : comportements abstraits
      - méthode `get2` retirant deux éléments d'un tampon borné -> oblige à subdiviser le comportement abstrait `partial` en deux sous-comportements : `one` et `partial`
- directions :
  - spécifications plus abstraites [McHale 94]
  - séparation entre synchronisation comportementale et intra-objet [Thomas PARLE'92]

Limite 2 : Duplication des invocations

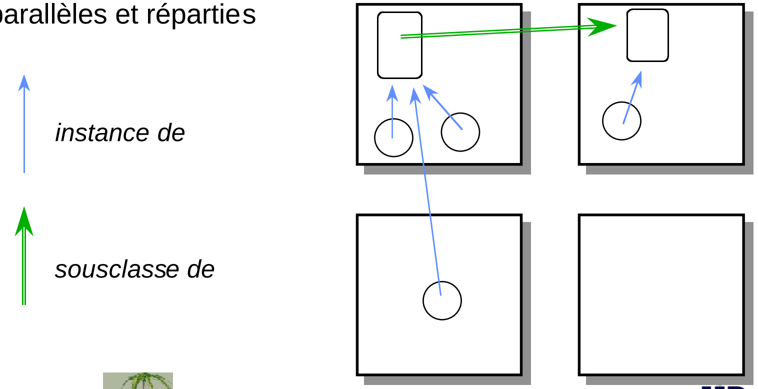
- Application directe des protocoles de duplication de serveurs (pour gérer la tolérance aux pannes) aux objets
  - *PROBLEME* : Ces protocoles font l'hypothèse qu'un serveur restera toujours un serveur simple (i.e., n'invoquera pas d'autres serveurs en tant que client)
  - <u>Cette hypothèse ne tient plus dans le monde objet...</u>
  - Si le serveur dupliqué invoque à son tour un autre objet, cette invocation sera dupliquée. Ce qui peut conduire à des incohérences



  - Solution possible : *pré-filtrage* par un coordinateur arbitrairement désigné (un des serveurs dupliqué) (en fait solution un peu plus complexe pour résistance aux pannes du coordinateur -> *post-filtrage*) [Mazouni et al. TOOLS-Europe'95]

Limite 3 : factorisation vs répartition

- Les stratégies standard de mise en œuvre (implémentation) des concepts d'objet (factorisation : classe et héritage) ont fait des hypothèses FORTES (séquentialité et mémoire centralisée)
  - Lien instance - classe
  - Lien classe - surclasse
- Elles ne peuvent être transposées directement à des architectures parallèles et réparties

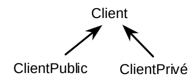


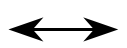
factorisation vs répartition (2)

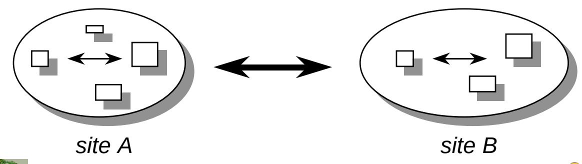
- Solution1 : dupliquer l'ensemble des classes
  - Cela suppose qu'elles sont immutables
    - » constantes de classe OK, mais pas de variables de classe
    - » <u>problème de mise à l'échelle</u>
- Solution2 : partitionner statiquement les classes en modules [Gransart'95]
  - Mais complexifie les possibilités de migration

factorisation vs répartition (3)

- Solution2' : méthodologie de partitionnement plus fine [Purao et al., CACM'8/98]
  - reconception d'une application existante



  - méthode semi-automatique
  - environnement aide et réalise les choix qui restent a la charge de l'utilisateur
  - modèle d'architecture : hiérarchique (à clusters)
  - distinction entre :
    - » communication inter-sites - grand coût 
    - » communication inter-processeurs (intra-site) - faible coût 



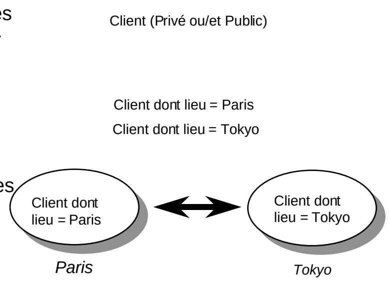
factorisation vs répartition (4)

phase 1 : répartition entre les sites

» refactorisation «roll up» des attributs/méthodes de sous-classes dans une super-classe pour éviter que l'héritage ne «traverse» PLUSIEURS sites

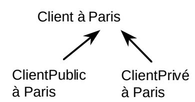
» puis fragmentation (spécialisation) des classes
  - à partir de scénarios d'interaction

» allocation des fragments (= sous ensembles d'instances) sur les différents sites
  - critère : minimiser les communications inter-sites

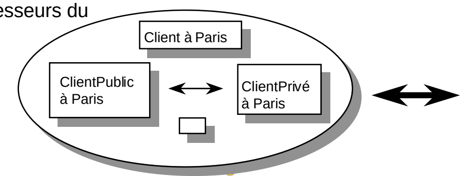


– phase 2 : répartition à l'intérieur d'un site donné

» redépliement de l'héritage «roll down» des fragments

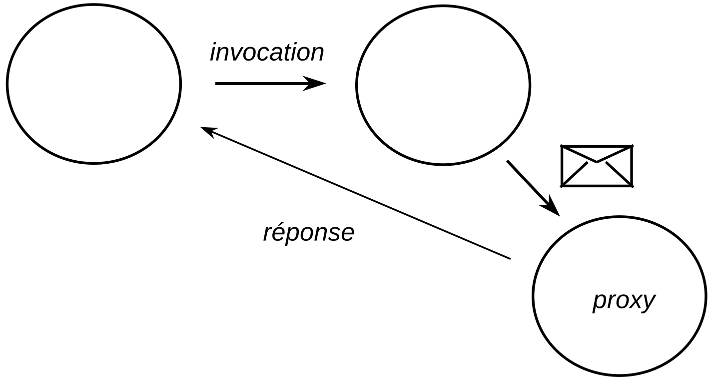


» optimisation de l'allocation des fragments sur les processeurs du site
  - décision multi-critères :
    – adéquation du processeur
    – concurrence
    – communication inter-processeurs
    – réplication des instances



Délégation

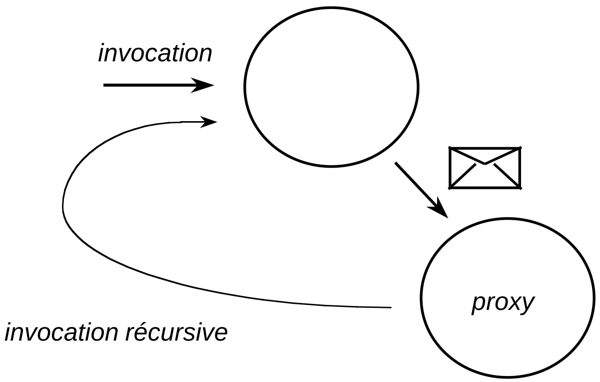
- Le mécanisme de délégation [Lieberman OOCP'87] offre une alternative a priori séduisante à l'héritage
  – Repose uniquement sur la transmission de messages, donc indépendant d'une hypothèse de mémoire centralisée



Délégation (2)

- *PROBLEME* : ordonnancement correct des invocations récursives, qui doivent être traitées AVANT les autres invocations

  -> synchronisation non triviale



Bilan

- Approche intégrée séduisante
  – nombre minimal de concepts
  – cadre unique
- Mais problèmes dans certains secteurs d'intégration
- Uniformité peut-être trop réductrice
  – Limites de la transparence et donc du contrôle
  – Problèmes d'efficacité
    » tout objet est actif
    » toute transmission de messages est une transaction
- Réutilisation des programmes standards/séquentiels existants
  – Identifier les activités et les encapsuler dans des objets actifs
  – Règles de cohabitation entre objets actifs et objets passifs

Méta-Bilan

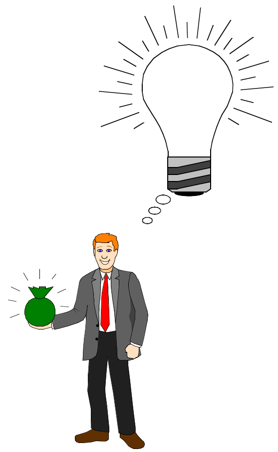
- Objectif : réconcilier le meilleur de l'approche applicative et de l'approche intégrée
- Observation : l'approche applicative et l'approche intégrative ne sont pas au même niveau
  - approche applicative pour le concepteur
  - approch intégrative pour l'utilisateur
- Comment interfacer des bibliothèques de composants et de protocoles destinées au concepteur (approche applicative)

  avec un langage uniforme destiné à l'utilisateur (approche intégrée)??



Réflexion

- Le concept de *réflexion* (méta-programmation, architectures réflexives...) offre justement un cadre conceptuel permettant un découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre*

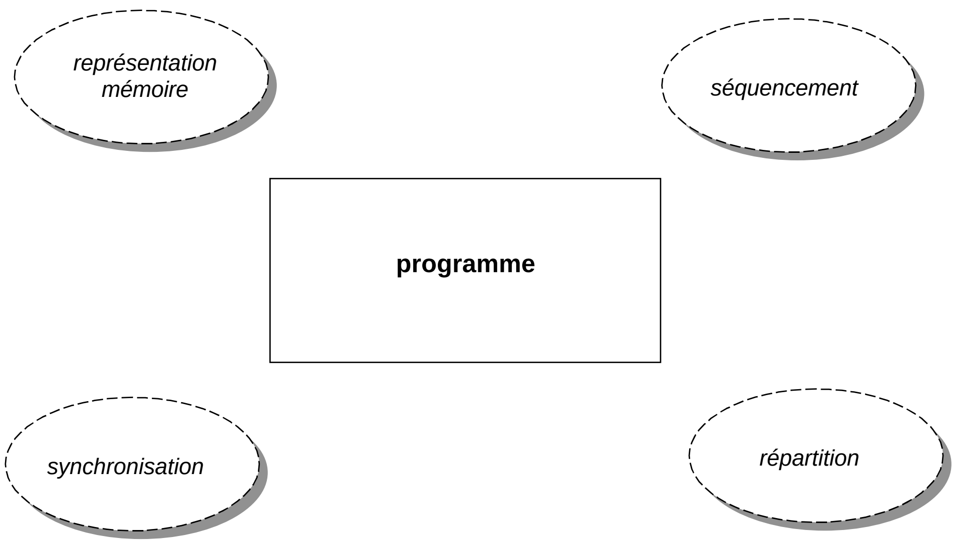


Réflexion (2)

- Diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes (*réifiées*) sous la forme de *méta-programmes*.
  - Habituellement elles sont invisibles et immuables (interprète, compilateur, moniteur d'exécution...)
- La spécialisation de ces méta-programmes permet de *particulariser* (éventuellement dynamiquement) l'exécution d'un programme
  - » représentation mémoire
  - » modèle de calcul
  - » contrôle de concurrence
  - » séquencement
  - » gestion des ressources
  - » protocoles (ex : résistance aux pannes)

  avec le minimum d'impact sur le programme lui-même

Contexte d'exécution



(Retour à un vieux) Dilemne

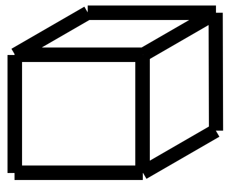
- Ecrire de BEAUX programmes
  - lisibles
  - concis
  - modulaires
  - abstraits
  - génériques
  - réutilisables
- Ecrire des programmes EFFICACES
  - spécialisés
  - choix optimaux de représentation interne des données
  - contrôle optimisé
  - gestion des ressources adéquate
- DILEMNE : Spécialiser/optimiser des programmes tout en les gardant génériques

Boîte noire

Modèle

Programme

exécuté par



Boîte noire

Mise en oeuvre (exécution) du programme est non modifiable

Interprète / Compilateur / Moniteur

Solutions Ad-Hoc

- Coder "entre" les lignes
  - difficile à comprendre
  - difficile à maintenir (hypothèses cachées)
  - peu réutilisable
- Annotations/Directives (déjà mieux)
  - ex : High Performance Fortran (HPF)
  - mais
    - » notations de plus ou moins bas niveau
    - » ensemble/effet des annotations non extensible/adaptable

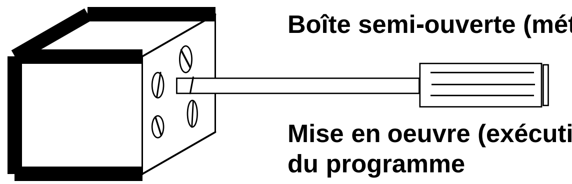
Réflexion (3)

Modèle

Programme

exécuté par

Boîte semi-ouverte (méta-interfaces)

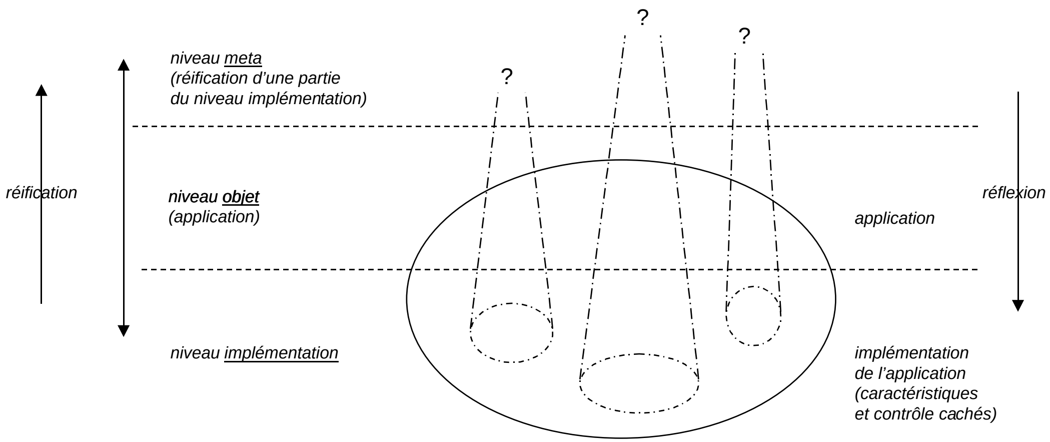


Mise en oeuvre (exécution) du programme est adaptable/spécialisable

Interprète / Compilateur / Moniteur

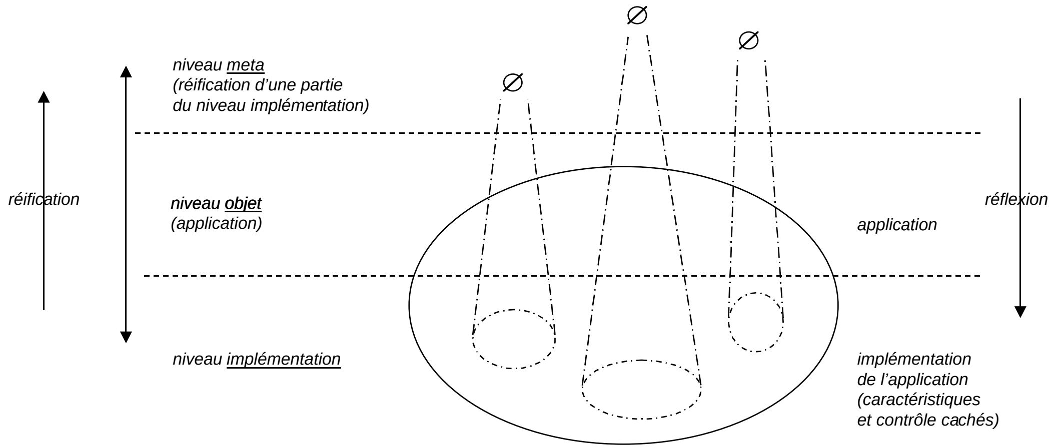
Open Implementation [Kiczales 94]

Réification/réflexion



Réification/réflexion

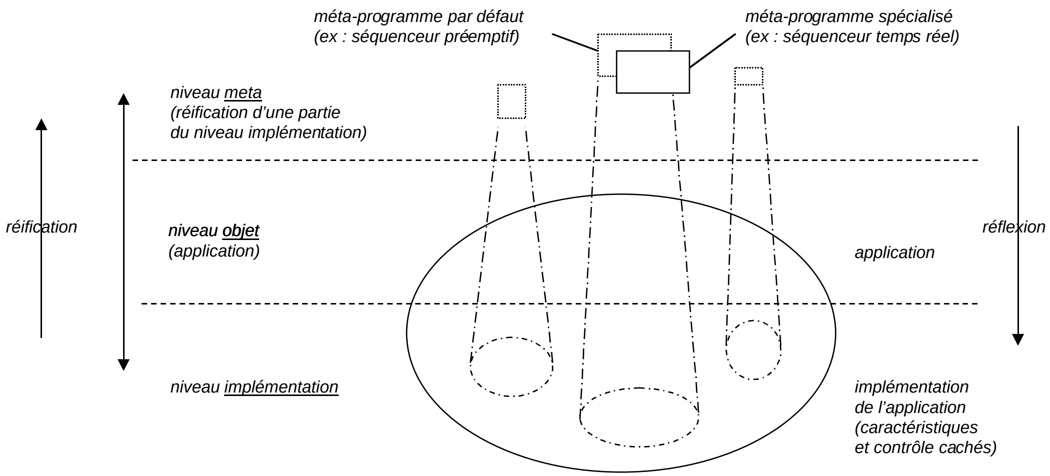
- Réification numérique (potentiomètres)
  - Ex : Options de compilation d'un compilateur
    - <u>Efficacité vs taille</u> du code généré



Réification/réflexion (2)

- Réification logicielle (méta-programmes)
  - Ex : algorithme de séquencement (scheduler)

plus général/flexible que des potentiomètres



Réflexion (4)

- Découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre (exécution)*
- Séparation entre programme ET méta-programme(s) favorise :
  - généricité et réutilisation des programmes
  - et des méta-programmes
- Ex :
  - changer la stratégie de contrôle pour un programme donné
  - réutiliser une stratégie de contrôle en l'appliquant à un autre programme

Structure / Dynamique

- Structure (représentation)
  - spécialiser la création des données
    - méthodes de classe (= méthodes de métaclasses) en Smalltalk
    - constructor member functions en C++, en Java
  - spécialiser un gestionnaire de fenêtres
    - implantation d'une feuille de calcul en Silica [Rao]
  - introspection
    - représentation d'entités du langage Java (ex : entiers, classes) sous forme d'objets Java
- Dynamique (comportement/exécution)
  - implémenter des coroutines via la manipulation de continuations
    - call/cc en Scheme
  - spécialiser le traitement d'erreur
    - doesNotUnderstand: en Smalltalk
  - changer l'ordre de déclenchement de règles de production
    - méta-règles en NéOpus

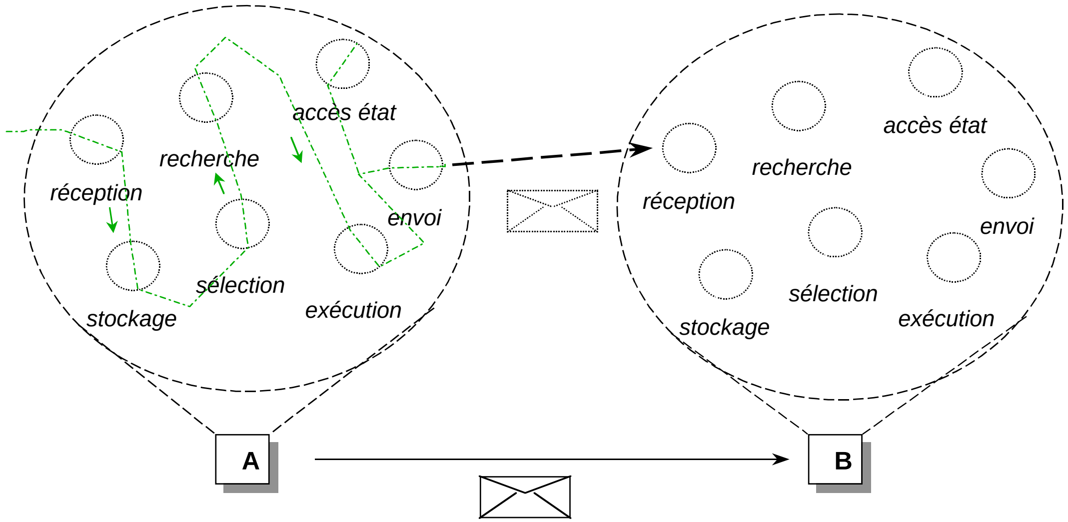
Approche réflexive

- Réflexion permet d'intégrer intimement des (méta-)bibliothèques de contrôle avec un langage/système
- Offre ainsi un cadre d'interface entre approche applicative et approche intégrée
- La réflexion s'exprime particulièrement bien dans un modèle objet
  - modularité des effets
  - encapsulation des niveaux
- méta-objet(s) au niveau d'un seul objet
- méta-objets plus globaux (ressources partagées : séquencement, équilibre de charges...)
  - *group-based reflection* [Watanabe'90]

Méta-objets/composants

- *CodA* [McAffer ECOOP'95] est un exemple de modèle relativement général d'architecture réflexive
- Sept méta-objets/composants de base :
  - envoi de message
  - réception de messages
  - stockage des messages reçus
  - sélection du premier message à traiter
  - recherche de méthode correspondant au message
  - exécution de la méthode
  - accès à l'état de l'objet
- Les méta-composants sont :
  - spécialisables
  - (relativement) combinables

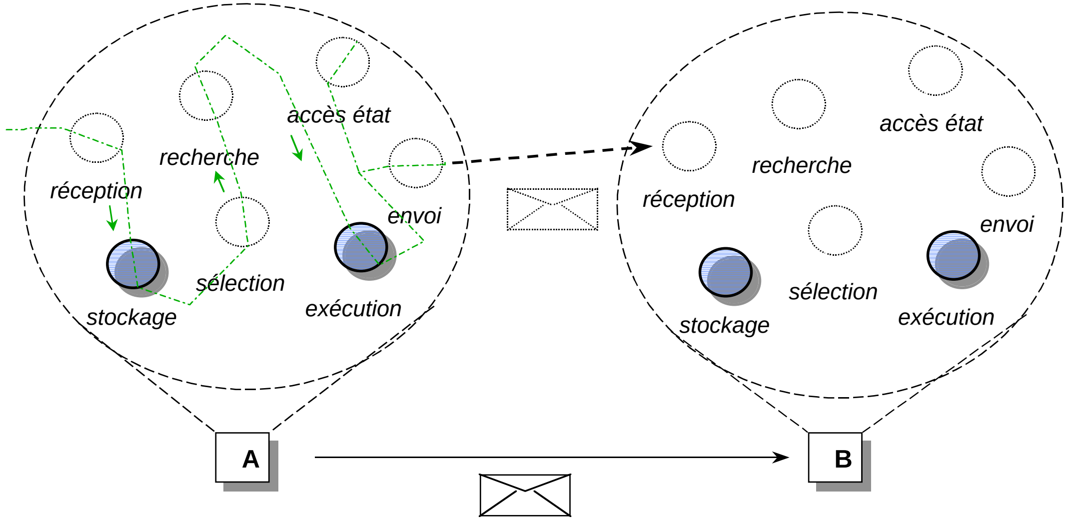
CodA



Ex : Exécution concurrente

– envoi de message
– réception de messages
– **stockage des messages reçus**
  » file d'attente (FIFO)
– sélection du premier message à traiter
– recherche de méthode correspondant au message
– **exécution de la méthode**
  » processus associé
  » boucle infinie de sélection et traitement du premier message
– accès à l'état de l'objet

Exécution concurrente (2)

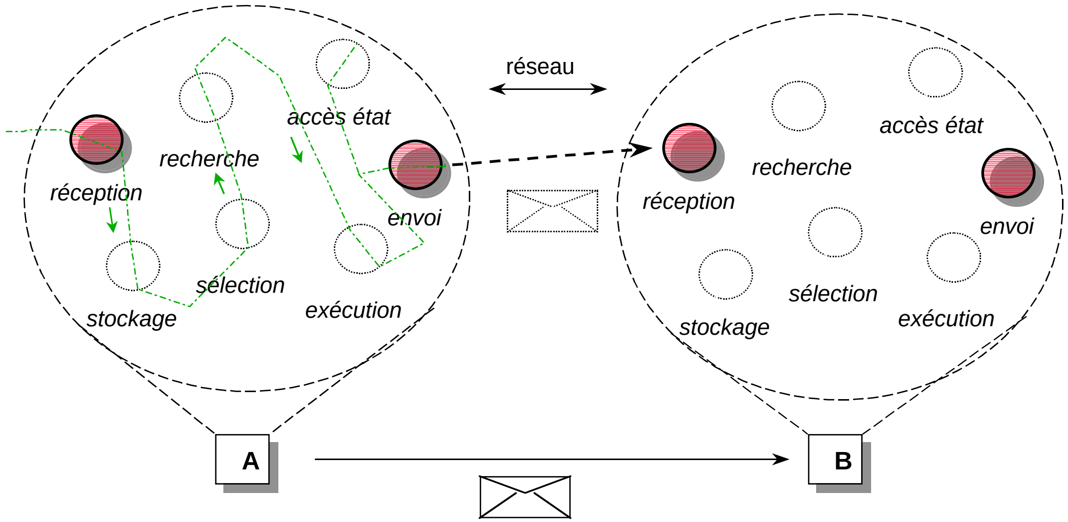


Ex : Exécution répartie

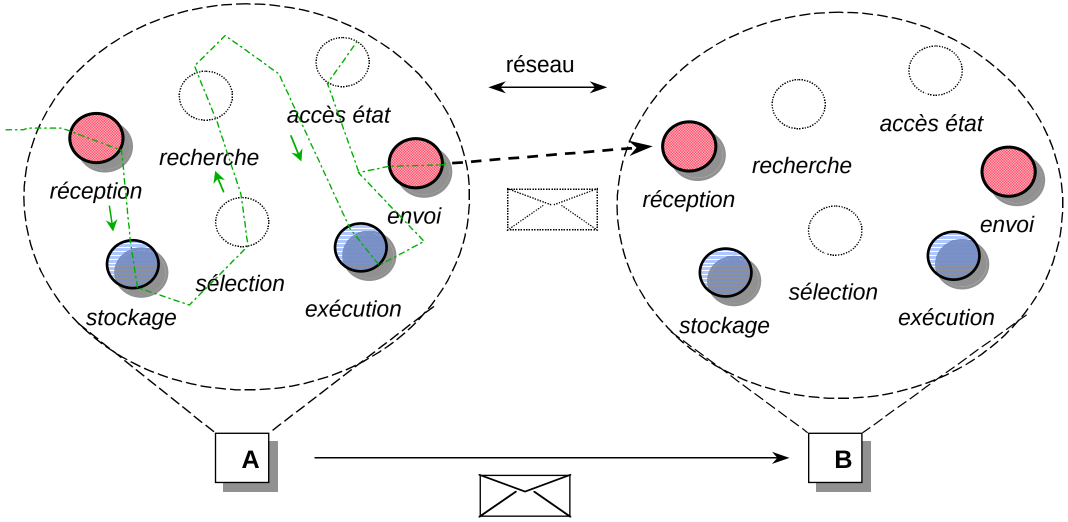
– **envoi de message**
  » encodage des messages, envoi via le réseau
– **réception de messages**
  » réception via le réseau, décodage des messages
– stockage des messages reçus
– sélection du premier message à traiter
– recherche de méthode correspondant au message
– exécution de la méthode
– accès à l'état de l'objet

– **encodage**
  » discipline d'encodage (marshal/unmarshal)
– **référence distante**
– **espace mémoire**

Exécution répartie (2)



Exécution concurrente et répartie (composition)

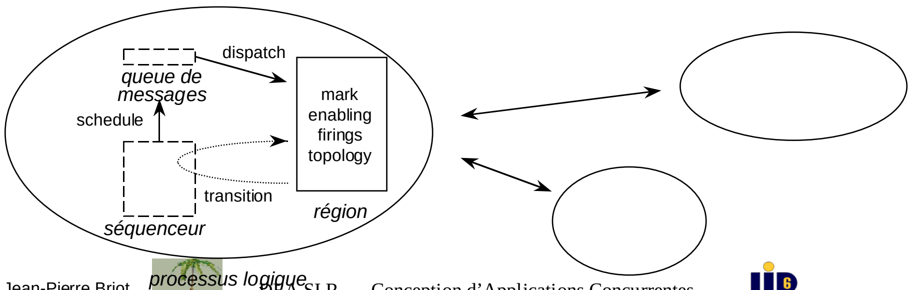


Méta-composants de plus gros grain (autres ex.)

- *Actalk* [Briot, LMO'94]
  - » activité / synchronisation (concurrence intra, acceptation explicite, gardes, compteurs de synchro...)
  - » communication (synchrone, asynchrone, future...)
  - » invocation (estampillage temporel, priorités...)
- *GARF* [Garbinato et al. 94]
  - » objet (persistant, dupliqué...)
  - » communication (multicast, atomique...)
  - » *ex : résitance aux pannes*
- *MAUD* [Agha et al. DCCA'93]
  - » envoi des messages
  - » réception des messages
  - » état
  - » *ex : installation dynamique de protocoles (duplication de serveurs, atomicité)*
- *AL1/D* [Okamura ECOOP'94]
  - » *ex : contrôle de la migration*

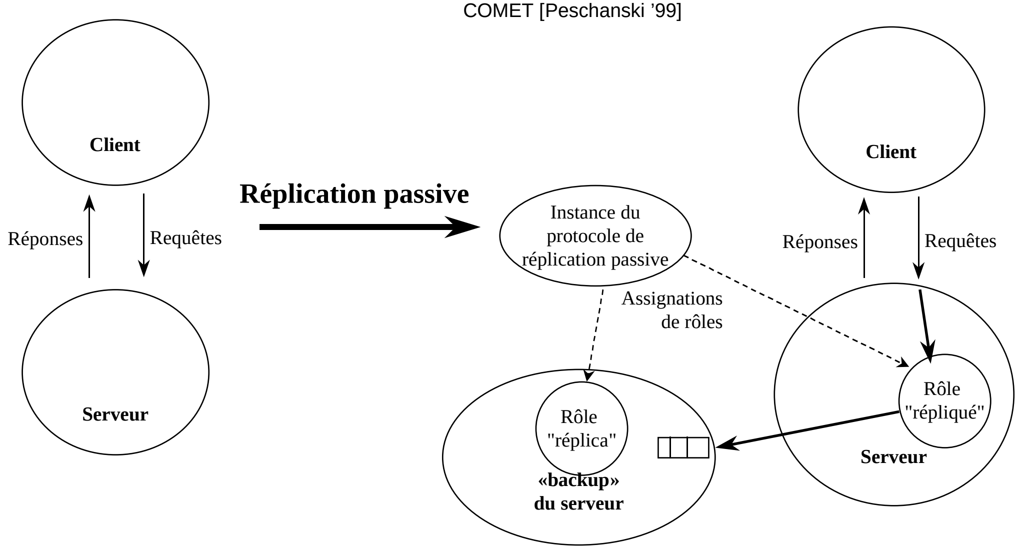
Temps réel

- méta-acteurs associés à des acteurs
  - – contrôle du temps pour du «soft real time» [Honda'92]
- machine de contrôle [Nigro et al., FMOODS'97] pour un ensemble d'acteurs
  - – méta-composants :
    - » horloge, queue de messages (= liste d'événements), contrôleur (période de simulation), séquenceur
    - » permettent de modifier les aspects temporels de l'application indépendamment de l'application elle-même
    - » ex : simulation distribuée optimiste (Time Warp) de réseaux de Petri temporels (timed Petri nets)

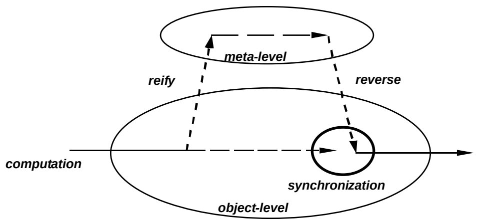


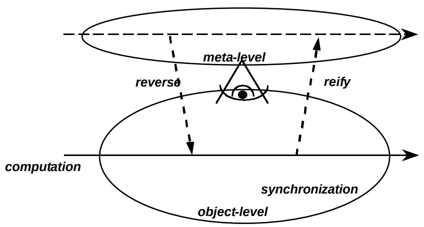
Réflexion sur un ensemble d'objets
Ex : Installation d'un protocole de réplication passive

COMET [Peschanski '99]



Réflexion dans les systèmes concurrents





Systèmes commerciaux

- *Muse (ex-Aperios)* [Yokote OOPSLA'92]
  - spécialisation dynamique de la politique de séquencement (ex : passer au temps réel)
  - application au «video on demand» et aux robots-chiens Aibo (Sony)
- Moniteur de transaction [Barga et Pu '95]
  - Incorporation de protocoles transactionnels étendus (relâchant certaines des propriétés standard : ACID)
  - dans un système existant
  - réification a posteriori via des upcalls
    - » (délégation de verrou, identification de dépendances, définition de conflits)

Bilan - Conclusion

- Approche réflexive prometteuse
- Architectures réflexives encore plus ou moins complexes, mais méthodologie s'établit et s'affine
- Validations en vraie grandeur en cours
- Retour du problème clé de la composition arbitraire (de méta-composants)
- (In)Efficacité
  - réduction de la portée de la réflexion (compilation)
    - » ex : OpenC++ version 2 [Chiba, OOPSLA'95]
  - transformation de programmes - évaluation partielle
    - » [Masuhara et al., OOPSLA'95]
- Ne dispense pas du travail nécessaire à l'identification des bonnes abstractions

Exemple : CORBA

- approche applicative :
  - structuration en bibliothèques
    - » services (ex : nommage, événements, transactions..)
    - » facilités (ex : interface utilisateur, gestion de tâches...)
    - » domaine d'application
- approche intégrative
  - objet distribué
    - » intégration transmission de message avec invocation distante
    - » transparence pour l'utilisateur
- approche réflexive
  - réification de certaines caractéristiques de la communication
    - » ex : smart proxies de Orbix (IONA)
      - ex d'utilisation : implantation de transmission de messages asynchrone
      - intégration des services avec la communication distante

II - Composants

(Limites) des objets...

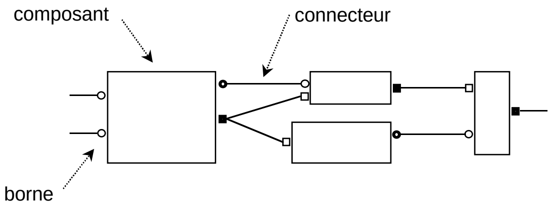
- granularité encore trop fine-moyenne
  - pas trop bien adapté à la programmation à grande échelle
- pas encore assez modulaire
  - références directes entre objets
  - donc connexion non reconfigurable sans changer l'intérieur de l'objet
    - » objet appelé
    - » nom de la méthode appelée



... aux composants

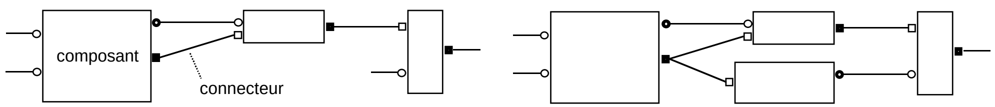
Idées :

- <u>composants</u>
  - plus «gros»
  - plus autonomes et encapsulés
  - <u>symétrie retrouvée</u> : interfaces d'entrée<u>S</u> mais aussi de <u>sorties</u>
  - plusieurs interfaces de sortie, et d'entrées : notions de "bornes"
- réification des relations/connexions entre composants
  - références hors des objets -> couplage externe (mais reste explicite)
  - notion de <u>connecteur</u>



Architectures logicielles

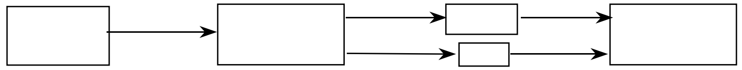
- Programmation à grande échelle
- Configuration et reconfiguration d'applications modulaires/réparties
- Composants
  - clients, serveurs, filtres, couches...
- Connecteurs
  - appels de procédure, messages, diffusion d'événements, pipes&filters...



Architectures logicielles [Shaw et Garlan 96]

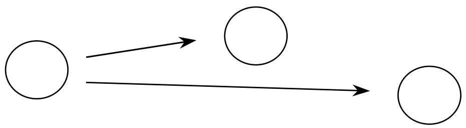
- différents types d'architectures (styles architecturaux)
  - pipes & filters, ex : Unix Shell `dvips | lpr`
  - couches, ex : Xinu, protocoles réseaux
  - événements (publish/subscribe), ex : Java Beans
  - frameworks, ex : Smalltalk MVC
  - repositories, ex : Linda, blackboards
- un même (gros) système peut être organisé selon plusieurs architectures
- les objets se marient relativement bien avec ces différentes architectures logicielles
  - objets et messages comme support d'implémentation des composants et aussi des connecteurs
  - cohabitation, ex : messages *et* événements

Ex1 : Pipes & filters



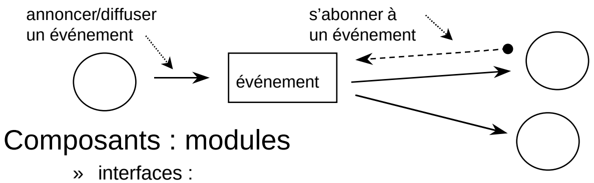
- Composants : filters
- Connecteurs : pipes
- Ex : Unix shell `dvips | lpr`
- +
  - » compositionalité (pipeline)
  - » réutilisabilité
  - » extensibilité
  - » analyses possibles (débit, deadlock...)
  - » concurrent
- -
  - » «batch», pas adéquat pour systèmes interactifs, ex : interfaces homme-machine
  - » petit dénominateur commun pour la transmissiond e données
    - performance
    - complexité

Ex2 : Objets & messages



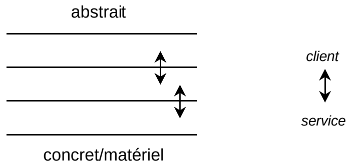
- Composants : objets
- Connecteurs : transmission de messages
- Ex : Java
- +
  - » encapsulation
  - » décomposition
- -
  - » références directes
    - nécessité de recoder les références si reconfiguration

Ex3 : Diffusion d'événements (publish/subscribe)



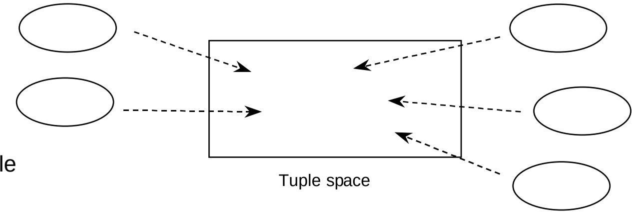
- Composants : modules
  - » interfaces :
    - procédures
    - événements
- Connecteurs : diffusion d'événements
- Ex : interfaces homme machine, bases de données (contraintes d'intégrité), Java Beans
- +
  - » réutilisation
  - » évolution
- -
  - » contrôle externe aux composants
    - difficile de déterminer quels modules seront activés, dans quel ordre...
  - » validation difficile

Ex4 : Systèmes en couches (layered systems)



- Composants : couches
- Connecteurs : appels de procédures
- Ex : protocoles de communication/réseaux, bases de données, systèmes d'exploitation (ex : Unix)
- +
  - » niveaux croissants d'abstraction
  - » extensibilité
  - » réutilisabilité
- -
  - » pas universel
  - » pas toujours aisé de déterminer les bons niveaux d'abstraction
  - » performance

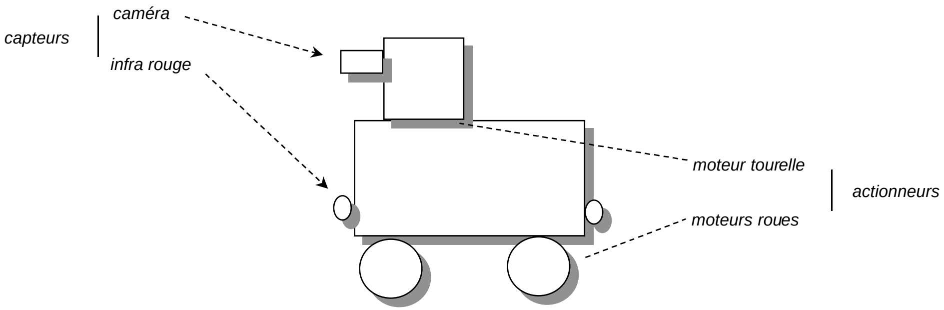
Ex4 : Repositories



- Composants :
  - structure de données centrale
  - processus
- Connecteurs : accès directs processus <-> structure
  - processus -> structure, ex : bases de données
  - structure -> processus, ex : démons, data-driven/trigger
- Ex : (Linda)Tuple space, blackboard (tableau noir)
- +
  - » partage des données
- -
  - » contrôle opportuniste

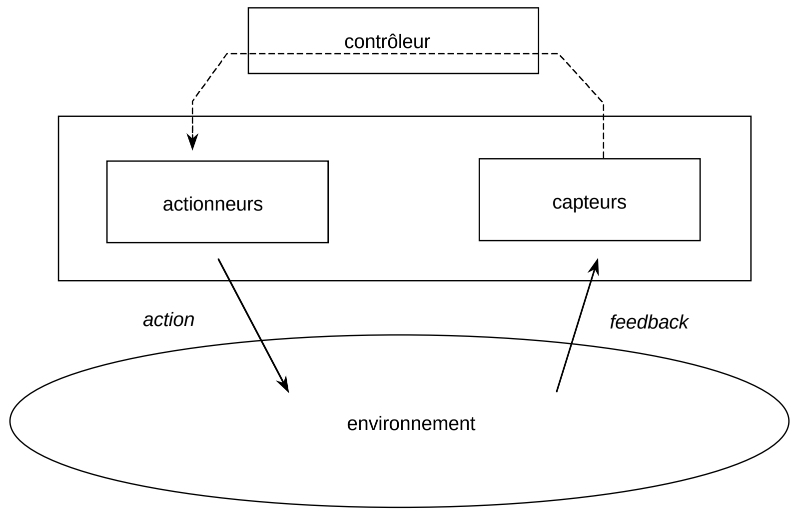
Comparaison de styles architecturaux

- Exemple d'application :
  - (architecture de contrôle d'un) robot mobile autonome

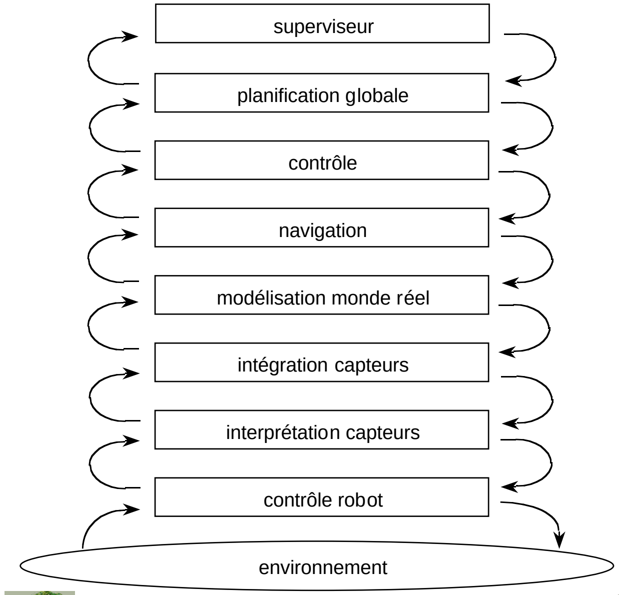


- Propriétés/caractéristiques recherchées :
  - comportement à la fois délibératif et réactif
  - perception incertaine de l'environnement
  - robustesse (résistance aux pannes et aux dangers)
  - flexibilité de conception (boucle conception/évaluation)

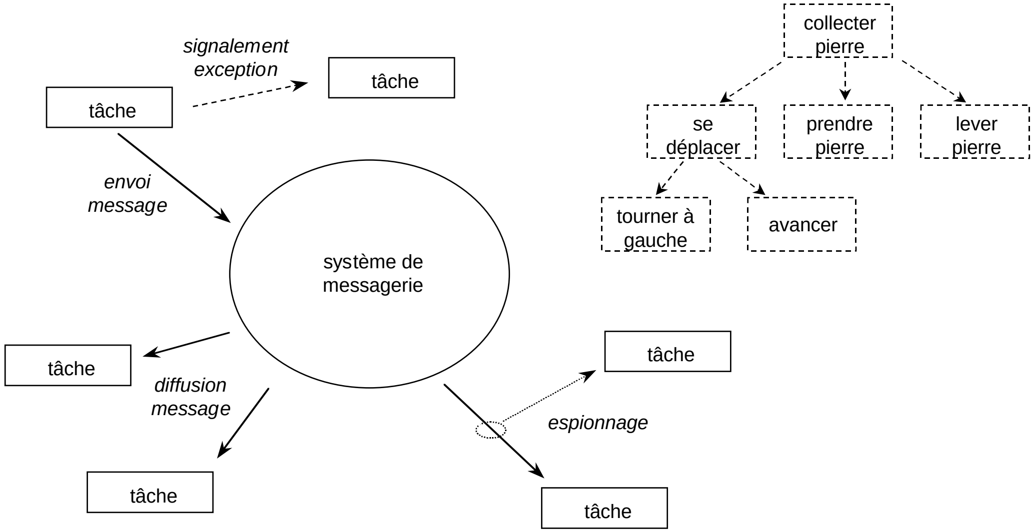
Solution 1 - boucle de contrôle



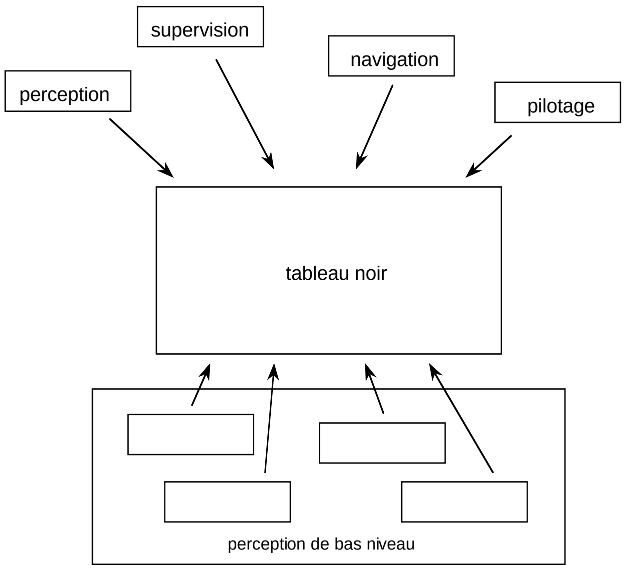
Solution 2 - couches



Solution 3 - (tâches et) événements



Solution 4 - tableau noir

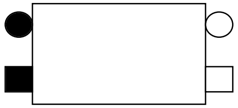


Comparaison

	Boucle de contrôle	**couches**	**événements**	**tableau noir**
coordination des tâches	**+ -**	**-**	**+ +**	**+**
incertain	**-**	**+ -**	**+ -**	**+**
robustesse	**+ -**	**+ -**	**+ +**	**+**
sûreté	**+ -**	**+ -**	**+ +**	**+**
performance	**+ -**	**+ -**	**+ +**	**+**
flexibilité	**+ -**	**-**	**+**	**+**

Langages de description d'architectures

- Architecture Description Languages (ADLs)
  - définition des composants
    - deux types d 'interfaces :
      - requises (in)
      - fournies (out)
    - sémantique d'appel
      - synchrone
      - asynchrone/événement
  - définition des connexions
    - connecteurs utilisés (ex : RPC)
  - vérification de la sémantique d'assemblage
    - conformité types/interfaces
    - contraintes de déploiement (OLAN)
      - ex : taille mémoire minimale, charge machines, etc.
  - Unicon [Shaw et al. 95]
  - Rapide [Lucham & Vera 95]
  - OLAN [Belissard et al. 96]

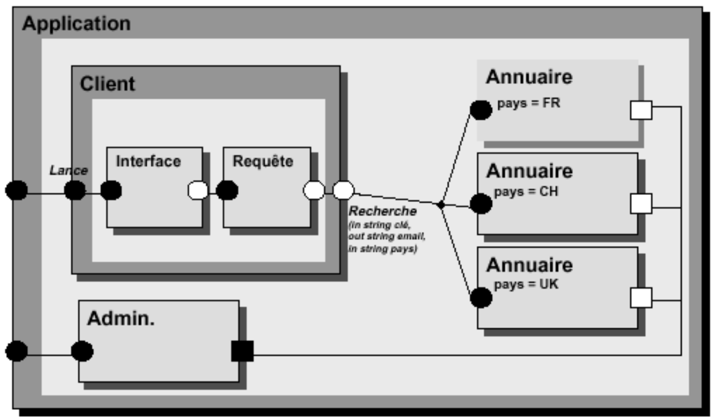


plus sur les ADLs (et le reste)

- Transparents de cours Ecole d'Eté sur la Construction d'Applications Réparties IMAG-INRIA-LIFL
- http://sirac.imag.fr/ecole/
  - 1998
  - 1999
- En particulier sur les ADLs (exemples en Unicon, OLAN, Rapide...) :
  - http://sirac.imag.fr/ecole/98/cours/composants.pdf
  - transparents de Michel Riveill
  - pages 3-4 et 27-43

  - également http://sirac.imag.fr/ecole/99/cours/99-8.pdf
  - et tous les autres transparents !
    - http://sirac.imag.fr/ecole/98/cours/
    - http://sirac.imag.fr/ecole/99/cours/

Exemple d'Architecture



6

transparent de Michel Riveill

UniCon

```
COMPONENT Annuaire
   INTERFACE IS
      TYPE Process
      PLAYER Lookup IS RPCDef
      SIGNATURE ("char *", "char**")
        End Lookup
   End INTERFACE

   IMPLEMENTATION IS
      VARIANT annuaire IS "annuaire.c"
      IMPLTYPE (Source)
   End IMPLEMENTATION
END Annuaire
```

```
COMPONENT Client
   INTERFACE IS
      TYPE Process
      PLAYER Lookup_Annuaire IS RPCCall
      SIGNATURE ("char *", "char **")
        End Lookup_Annuaire
   End INTERFACE

   IMPLEMENTATION IS
        VARIANT client IS "client.c"
        IMPLTYPE(source)
   END IMPLEMENTATION
End Client
```

57

transparent de Michel Riveill

UniCon

```
COMPONENT Application
   INTERFACE IS General
   // pas de typage strict utilisation du type
   générique
   END INTERFACE
   IMPLEMENTATION IS

// Définition des interfaces utilisées
    USE client1 INTERFACE Client
        PROCESSOR ("toua.inrialpes.fr")
        ENTRYPOINT (client1)
    END client1
    USE annuaire1  INTERFACE Annuaire
        PROCESSOR("dyade.inrialpes.fr)
    END annuaire1

// Définition des interfaces
    ESTABLISH Remote-proc-call WITH
        client1.Lookup_Annuaire AS calle
         annuaire1.Lookup AS definer
    END Remote-proc-call
   END IMPLEMENTATION
END Application
```

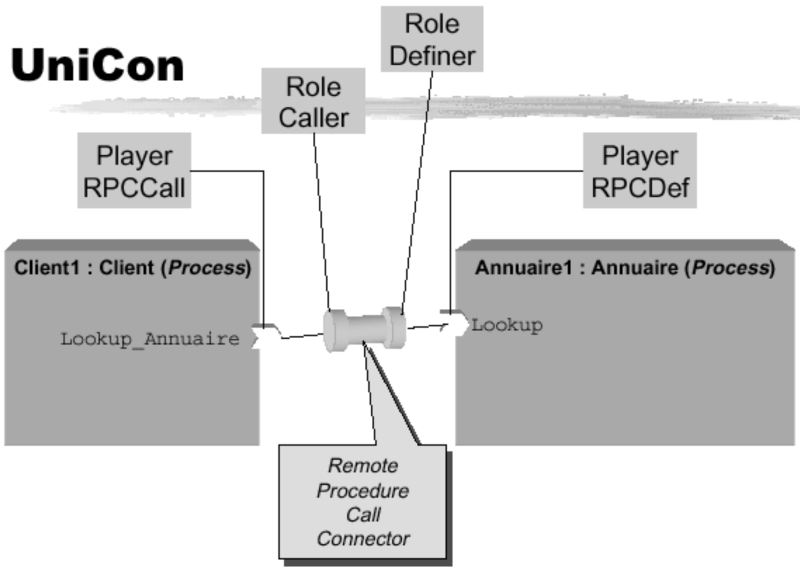
58

Unicon

```
CONNECTOR Remote-proc-call
   PROTOCOL IS
       TYPE RemoteProcCall
       ROLE definer IS definer
       ROLE caller IS caller
   END PROTOCOL
   IMPLEMENTATION IS
       BUILTIN
   END IMPLEMENTATION
END Remote-proc-call
```

59

UniCon



56

Composants

- Un composant est du code exécutable et son mode d 'emploi
  - module logiciel autonome (et persistant)
  - exporte interfaces
  - auto-description
  - « composable »
- Composants « source »
  - architectures logicielles
  - ex : Sun JavaBeans
- Composants binaires
  - ex : Microsoft COM
- « Petits » composants
  - ex : composants graphiques JavaBeans
- « Gros » composants
  - ex : MS Word, ILOG Solver...

Pourquoi les composants ? [Albert et Haren 2000]

- Analyse sur + de 2000 clients de composants (ILOG et autres)
  - 11 Critères pour l 'application développée (à base ou pas de composants) :
- flexibilité offerte *(éventail de choix ou forte rigidité)*
  - ex : fenêtres rondes rares et difficiles à intégrer
  - peut brider l'imagination des architectes
- compétences requises *(communes ou rares/pointues)*
  - conception vs utilisation
- moyens nécessaires au projet *(incluant déploiement et maintenance)*
  + coût de développement important, + composants avantageux
- vitesse de développement
  - excellente avec composants, ex : presque indispensable aux startups
  - mais adaptation composants peut être difficile
- incrémentalité du développement
  - porte sur l'extension de certains composants du prototype
- fiabilité du résultat
  - composants améliorent toujours fiabilité (capitalisation des tests)
  - mais (factorisation fait que la) criticité des composants augmente

Pourquoi les composants ? (2)

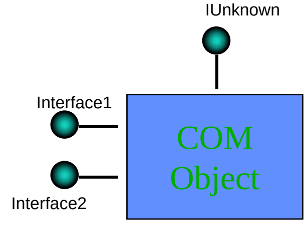
- performance du résultat final
  - performance en général inversement proportionnelle à généricité
  - mais capitalisation de l'optimisation
- facilité de déploiement *(portabilité sur différentes plates-formes)*
  - capitalisation des portages
  - utilisation quasi-générale pour les IHM
- indépendance vis-à-vis des fournisseurs *(possibilités de migrer d'un fournisseur à un autre, absorber la disparition ou rachat par compétiteur...)*
  - actuellement interfaces encore souvent propriétaires
  - pérennité du contrat avec fournisseurs de composants vs grand turnover développeurs internes
- lisibilité du code source
  - interne : découpage forcé en composants l'améliore
  - externe : API documentées facilite lisibilité du logiciel métier
- répétabilité du processus *(réutilisabilité code-source, savoir-faire, équipe...)*
  - capitalisation de l'apprentissage de l'utilisation de composants

COM / DCOM / ActiveX (d'après Peschanski&Meurisse)

- COM : ***Component Object Model***
- Définition d'un standard d'interopérabilité de **Composants binaires**
  - Indépendant du langage de programmation (i.e VB et C++ ?)
  - Modèle de composants extrêmement simple (voire vide...)
  - notion de composition de composants limité à la notion d'interface (*containment / aggregation*)
- But : fournir un modèle à base de composants le plus simple possible permettant l'adaptabilité, l'extensibilité, la transparence à la localisation (in process, local, remote) et des performances optimums...

Principes de COM (d'après Peschanski&Meurisse)

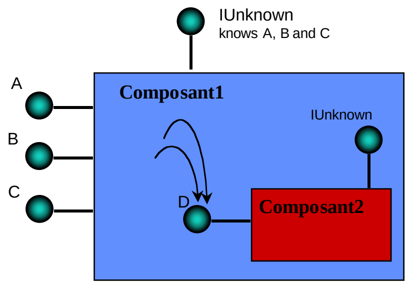
- Encapsulation "totale"
  - *Black-Box* : chaque composant est vu comme une boîte noire
  - L'interopérabilité entre composants ne se fait que via leurs *interfaces*
  - Possibilité de définir des *interfaces multiples* pour un même composant
  - QueryInterface : 'découvrir' les interfaces en cours d'exécution ***(réflexion !!)***
  - IUnknown : gestion du cycle de vie des composants (GC)

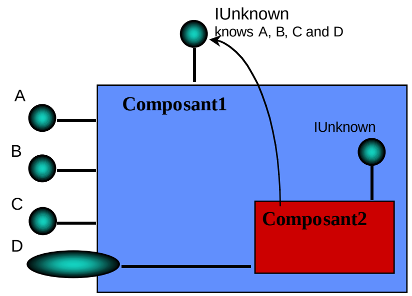


La composition dans COM (d'après Peschanski&Meurisse)

Principes de 'Réutilisabilité' [Microsoft97]

Par confinement / délégation **Par agrégation**



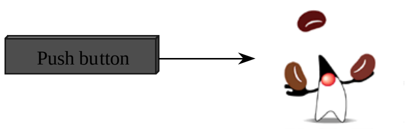


Cycle de vie des composants ('Versioning')...

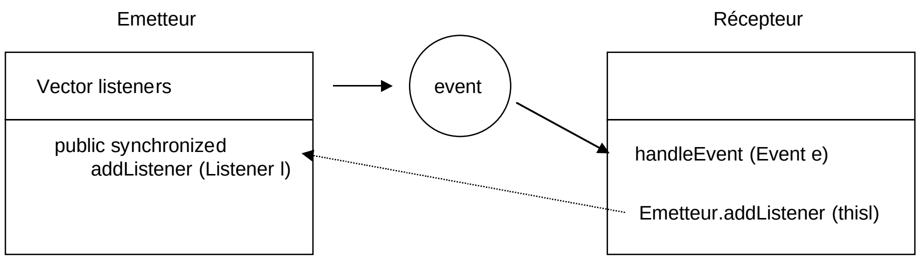
JavaBeans (d'après Peschanski&Meurisse)

- **Motivations** : Composition graphique d'applications
- Définition :
  - Entité logicielle manipulable graphiquement
  - **"A Java Bean is a reusable software component that can be manipulated visually in a builder tool." [Sun Spec97]**
- "**Modèle**" inspiré des *Architectures logicielles*
- mais principalement orienté **implémentation**...

Communication des JavaBeans



Inspiré d'un style architectural :
Communication implicite
(*publish/subscribe*)

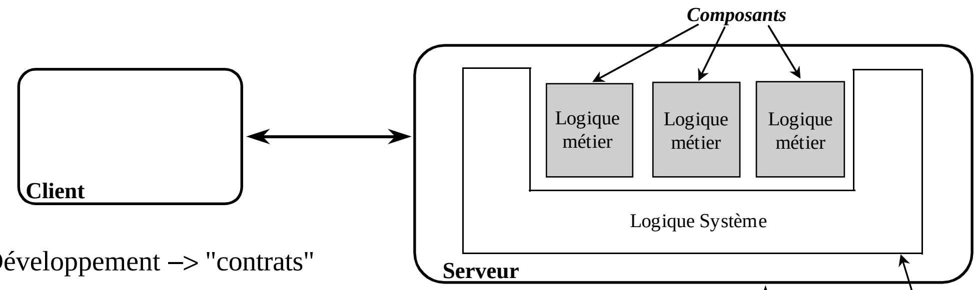


Propriétés JavaBeans (d'après Peschanski&Meurisse)

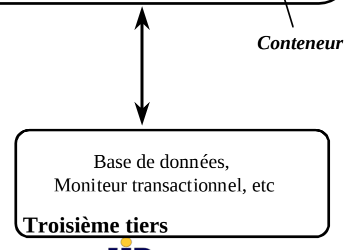
- Propriétés (méthodes *get* - *set*) - Editeurs de propriétés spécialisés (*Customizers*)
- Introspection granularité méthode/attribut
- Déploiement - Packaging (JAR)
- Support de Sérialisation Beans - Evénements
- etc.

Enterprise JavaBeans (d'après Peschanski&Meurisse)

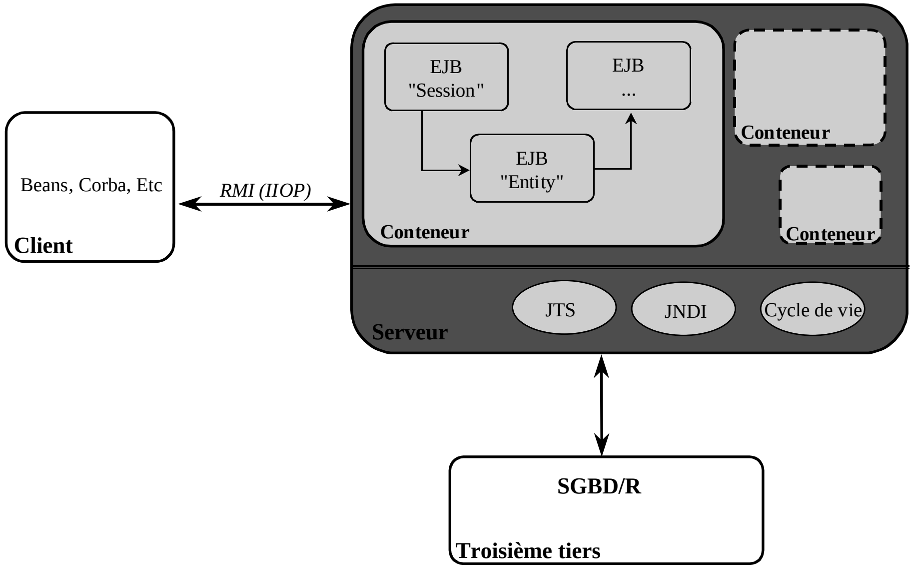
- But : **Simplifier** le développement d'architectures "3 tiers", côté serveur



- Développement –> "contrats"
- Déploiement
- Persistance
- Sécurité
- Transactions –> "déclaratif"
- Packaging –> "ejb-jar"



Exemple Entreprise JavaBeans (d'après Peschanski&Meurisse)



Composants Corba (d'après Peschanski&Meurisse)

Des objets (Corba 2.2) ⟶ **aux composants (Corba 3)**

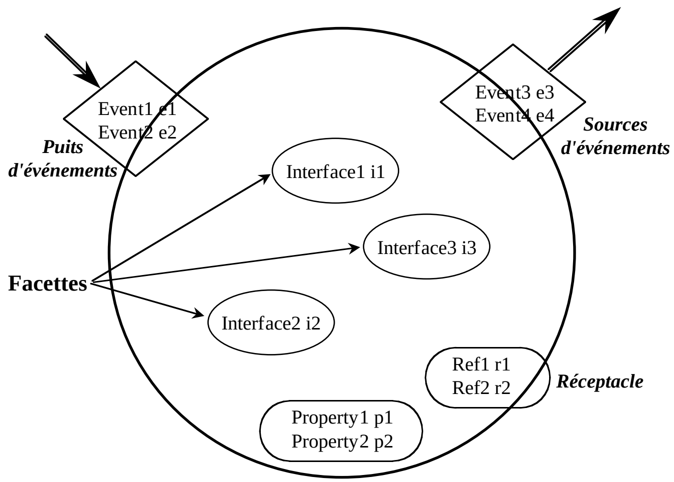
Serveurs archi client/Serveur — *Serveurs architecture 3 tiers*

EJB + interopérabilité (y compris avec les EJBs)

- Modèle abstrait –> IDL étendu
- Modèle de programmation –> CIDL + interfaces standards (API composant - conteneur)
- modèle d'exécution –> Conteneur + structures d'accueil + interfaces
- Modèle de déploiement –> Langage OSD (DTD XML) + interface
- meta-modèle –> MOF

Composants Corba : modèle abstrait (d'après Peschanski&Meurisse)

Modèle abstrait




```
component {
  attribute Property1 p1;
  attribute Property2 p2;

  provides Interface1 i1;
  provides Interface2 i2;
  provides Interface3 i3;

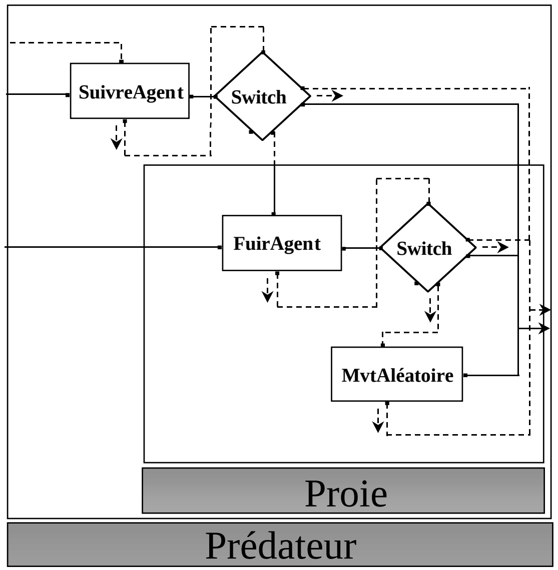
  consumes Event1 e1;
  consumes Event2 e2;

  uses Ref1 r1;
  uses multiple Ref2 r2;

  emits Event3 e3;
  publishes Event4 e4;
} exemple;
```

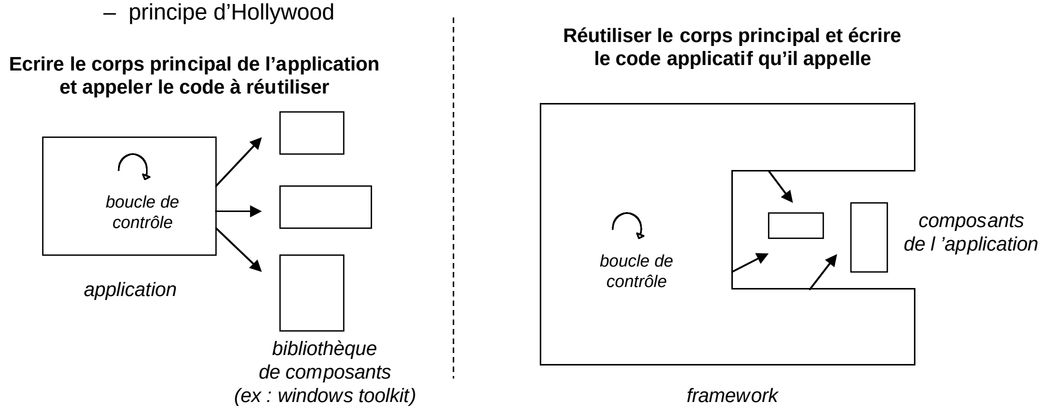
component exemple

Exemples de Conception



Frameworks

- Squelette d'application
- Ensemble de classes en collaboration
- Framework vs Boîte à outils (Toolkit)
  - inversion du contrôle
  - principe d'Hollywood



Frameworks (2)

- Un framework est une généralisation d'un ensemble d'applications
- Un framework est le résultat d'itérations
- Un framework est une unité de réutilisation
- Un framework représente la logique de collaboration d'un ensemble de composants : variables et internes/fixés

- « If it has not been tested, it does not work »

Corollaire :

- « Software that has not been reused is not reusable »
  [Ralph Johnson]

Exemples :

- Model View Controller (MVC) de Smalltalk
- Actalk

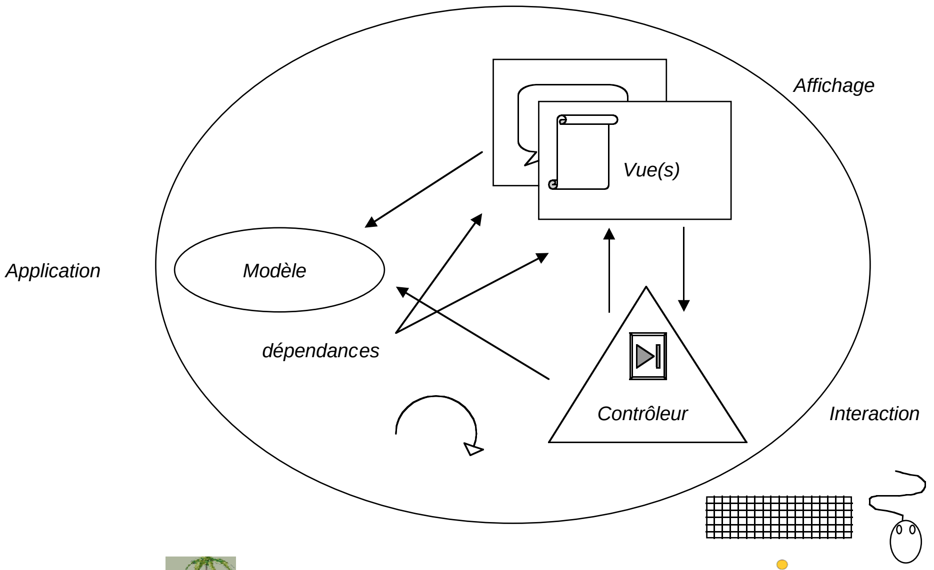
Architectures logicielles/Composants vs Frameworks

- Architectures logicielles et composants (et connecteurs)
  - Générique
  - Approches de conception
    - » descendante
      - décomposition
      - connexions
    - » ou ascendante
      - assemblage de composants existants
  - Les connexions et la coordination (« boucle de contrôle ») restent à définir, puisqu'elle est spécifique à l'application : difficile !

- Frameworks
  - Conception initiale du framework ascendante
  - Mais utilisation (spécialisation du framework) descendante
  - Les connexions et la coordination sont déjà définies (et testées) pour une classe d 'applications : plus facile !

Model View Controller (MVC)

Modèle d'interface homme-machine graphique de Smalltalk



Patrons de conception (Design Patterns)

- Idée : identifier les solutions récurrentes à des problèmes de conception
- « Patterns in solutions come from patterns in problems »
  [Ralph Johnson]
- Analogie :
  – principes d'architecture (bâtiments, cathédrales) [Christopher Alexander]
  – patrons/archétypes de romans (ex : héros tragique : Macbeth, Hamlet...)
  – cadences harmoniques : II-V-I, Anatole...
- Des architectes (C. Alexander) aux architectes logiciels
  – Design Patterns : Elements of Reusable O-O. Software
    [E. Gamma, R. Helm, R. Johnson, J. Vlissides (the « GoF »), Addison Wesley 1994]
- Les patterns ne font sens qu 'à ceux qui ont déjà rencontré le même problème (Effet « Ha ha ! »)
  – documentation vs génération

Pattern = < Problème , Solution >

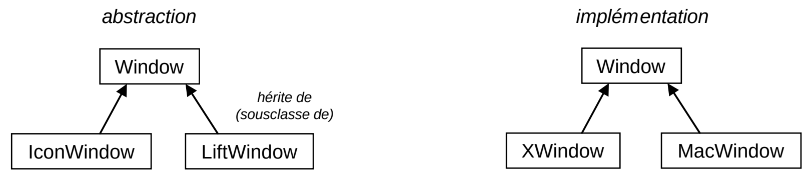
- Un pattern n'est *pas* juste une solution, mais est la discussion d'un type de solution en réponse à un type de problème
- nom (ex : Bridge, Observer, Strategy, Decorator...)
- contexte
- problème
- forces
- collaboration (possible avec d'autres patterns)
- directives
- exemples
- ...

Utilisation :

- Capitalisation de connaissances
- Explication
- Génération (vers une instanciation automatique de patterns)

Ex : pattern Bridge

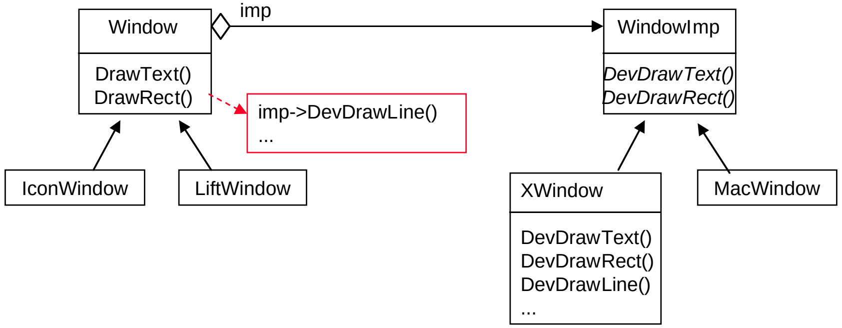
- Problème : une abstraction peut avoir différentes implémentations



- Solution naïve :
  – énumérer/nommer toutes les combinaisons
    - MacIconWindow, XIconWindow, etc.
  – problèmes :
    » combinatoire
    » le code du client dépend de l'implémentation

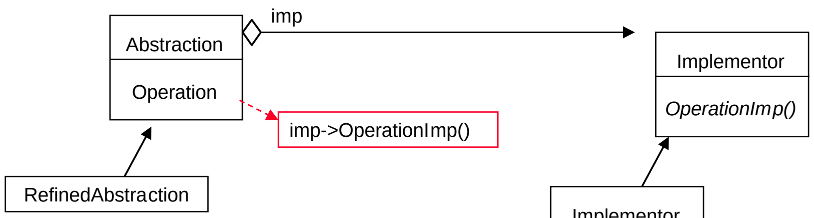
Ex : pattern Bridge (2)

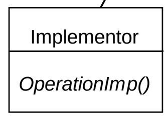
- Solution :
  - séparer les 2 hiérarchies



Ex : pattern Bridge (3)

- Solution générale (le pattern Bridge)





- Exemples d'utilisation :
  - Multi-fenêtres, libg++, Next AppKit...
- Egalement :
  - Langages de patterns (Pattern Languages), ex : GoF book
  - Patterns d'analyse (Analysis Patterns)
  - Patterns seminar group (équipe de Ralph Johnson à UIUC)
  - Voir : `http://www.hillside.net/patterns/patterns.html`
  - Crise actuelle des patterns ?

III - Agents

(sous)-Plan

- Première partie : Introduction aux Agents
  - Pourquoi les Agents ?
  - Positionnement historique (évolution de l'IA et de la Programmation)
  - Classification (incluant Agents Mobiles et Agents Assistants)
  - Principes
  - Architectures d'Agents

Motivations - pourquoi les agents?

- Complexité croissante des applications informatiques, plus ouvertes, plus hétérogènes, plus dynamiques
  - exemple : le Web et toutes les couches et services qui le supportent
  - comment décomposer, recomposer, interopérer, gérer l'évolution, adaptation (aux autres modules logiciels, à l'environnement, aux utilisateurs...), contrôle, négocier (partage ressources, prise de RdV),...
  - limitations des approches informatiques classiques : statiques, homogènes, interfaces rigides, objets/composants sans initiative propre, client serveur
  - difficile à maîtriser par des humains

Exemples

- Contrôle de sonde/vaisseau spatiale
  - Distance avec le contrôle au sol -> temps de réaction
  - -> Nécessité d 'un contrôle local : autonomie
  - capacités de prises de décision en cas de situations non prévues : initiative
- Recherche d'information sur Internet
  - Processus long et difficilement prédictible (attente, découverte, pannes…)
  - -> Délégation du cahier des charges : guidé par les objectifs
  - ex : recherche multilingue - coopération de différents agents
    - » (personnalisation, ontologie, dérivations, traduction, etc.) [Projet SAFIR]
- Prise de RdV
  - fastidieux, attentes (indisponibilité ou déconnexions)
  - -> PDAs assistants (apprend habitudes utilisateur et initiative) et coopératifs
- etc, ex : Surveillance de réseaux
  - détection, intervention, réparation

Idées

- Agents logiciels
  - autonomie
  - mission
  - initiative
  - niveau connaissance
  - adaptation
  - inter-opérabilité
  - De plus, ils peuvent être coopératifs (avec autres agents)
    - ex : prise de RdV distribuée
  - On parle alors de :
- Systèmes multi-agents
  (issus du domaine « résolution distribuée de problèmes »)
  - protocoles de communication
  - protocoles de coordination
  - organisations

Qu'est-ce qu'un agent ?

- Petit Robert :
  - De agere « agir, faire »
    - « Celui qui agit (opposé au patient qui subit l'action) »
    - « Ce qui agit, opère (force, corps, substance intervenant dans la production de certains phénomènes) »
  - De agens « celui qui fait, qui s 'occupe de »
    - « Personne chargée des affaires et des intérêts d'un individu, groupe ou pays, pour le compte desquels elle agit »
    - « Appellation de très nombreux employés de services publics ou d'entreprises privées, généralement appelés à servir d'intermédiaires entre la direction et les usagers »
- American Heritage Dictionary :
  - « one that acts or has the power or authority to act... or represent another »
  - « the means by which something is done or caused; instrument »

Qu’est-ce qu’un agent ? (2)

- [Ferber 95]
  - on appelle agent une entité physique ou virtuelle
    - qui est capable d’agir dans un environnement,
    - qui peut communiquer directement avec d’autres agents,
    - qui est mue par un ensemble de tendances (sous la forme d’objectifs individuels ou d’une fonction de satisfaction, voire de survie, qu’elle cherche à optimiser),
    - qui possède des ressources propres,
    - qui est capable de percevoir (mais de manière limitée) son environnement,
    - qui ne dispose que d’une représentation partielle de cet environnement (et éventuellement aucune),
    - qui possède des compétences et offre des services,
    - qui peut éventuellement se reproduire,
    - dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu’elle reçoit.

Rappel historique (vis à vis de l’IA)

- Concept d’agent rationnel à la base de l’intelligence artificielle (IA)
  - système informatique autonome
    - connaissances, buts, pouvoirs, perceptions, raisonnement/délibération (résolution, planification, déduction, etc.), actions
    - système expert
- Limitation : Autarcie !!
  - autarcie logicielle : difficile à faire collaborer avec d’autres logiciels
  - autarcie sociale : censé remplacer l’homme, pas de collaboration (expert humain en dehors de la « boucle »)
- Réponses
  - agents coopératifs
    - systèmes multi-agents
    - issus de la résolution distribuée de problèmes
      - distributed artificial intelligence (DAI versus GOFAI)
  - agents assistants

Rappel historique (vis à vis de la programmation)

- Interview Les Gasser, IEEE Concurrency 6(4):74-81, oct-déc 98

- langage machine
- assembleur
- programmation structurée
- programmation par objets
- programmation par agents !

- concept d’action persistante
- programme qui tente de manière répétée (persistante) d’accomplir quelque chose

- *mission et initiatives pour l’accomplir*

action persistante

- programme qui tente de manière répétée (persistante) d’accomplir quelque chose
  - pas la peine de contrôler explicitement succès, échec, répétition, alternatives...
- description de :
  - (quand) but == succès
  - méthodes alternatives
  - * apprentissage (de nouvelles méthodes)
- ressources :
  - processus
  - itération (tant que)
  - options/solutions (situation -> action)
  - capacité de choix (on line - sélection d’action)
  - recherche (search) -- en cas de nouvelles situations
  - feedback sur le choix

Une vision différente du logiciel (vers un couplage sémantique et adaptatif)

- Problème clé du logiciel : évolution, adaptation
  - profil utilisateur, programmeur, environnement, contraintes - ex : QoS, ...
  - Pour un système (logiciel) complexe, impossible de prédire au moment de la conception toutes les interactions potentielles
  - Ceci est rendu encore plus difficile si l'on considère l'évolutivité du logiciel ainsi que celle de son environnement (autres logiciels)
- Vers des composants logiciels « adaptables »
  - Les interactions non prévues deviennent la norme et non plus l'exception [Jennings 1999]
  - Le couplage entre composants est abordé au niveau des connaissances et non plus au niveau des types de données (ce qui est sûr mais rigide)
  - <u>Vers un plus grand découplage</u> : objets -> composants -> agents (et ensuite ?!)
  - A rapprocher du "Ever late binding" (C -> C++ -> Java -> ...)

typologie (Babel agents) 1/3

- agents rationnels
  - IA, comportement délibératif, perceptions, croyances, buts
  - ex : systèmes experts
- systèmes multi-agents
  - résolution distribuée (décentralisée) de problèmes
  - coordination, organisation
  - ex : robotique collective
- agents logiciels
  - ex : démons Unix, virus informatiques, robots Web
- agents mobiles
  - code mobile -> objet mobile -> agent mobile (processus)
  - motivations : minimisation communications distantes, informatique nomade
  - technologie en avance sur les besoins
  - problèmes de sécurité, coquilles vides

typologie (Babel agents) 2/3

- agents assistants
  - secrétaire virtuelle (trie le mail, gère les RdVs...)
  - < logiciel utilisateur + assistant >
  - filtrage collaboratif
    - ex : recommandation achats CDs par recherche de similarité des profils puis transitivité
  - computer-supported cooperative work -> communityware (pour citoyens)
  - agents « émotionnels »
- agents robotiques
  - architectures de contrôle de robots
  - sélection de l'action
  - robotique collective (ex : RoboCup, déminage...)
- vie artificielle
  - alternative à l'IA classique
  - modélisation/simulation des propriétés fondamentales de la vie (adaptation, reproduction, auto-organisation...)
  - importation de métaphores biologiques, éthologiques...
  - ex : algorithmes à base de fourmis (agents) pour routage de réseaux

typologie (Babel agents) 3/3

- simulation multi-agent
  - simulation centrée individu vs modèle global (ex : équations différentielles)
  - modèle de comportement arbitrairement complexe
  - interactions arbitrairement complexes (ex : sociales : irrigation parcelles)
  - niveaux hiérarchiques (ex : bancs de poissons)
  - espaces et échelles de temps hétérogènes
  - collaboration informaticien - spécialiste
- agents de loisir
  - virtuels (ex : jeux vidéo)
  - virtuels-physiques (ex : Tamagotchi)
  - physiques (ex : Furby, robot-chien Aibo de Sony)

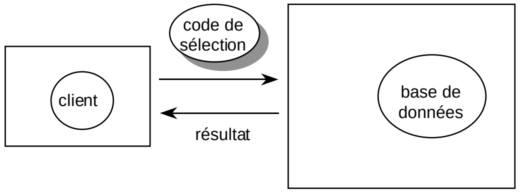




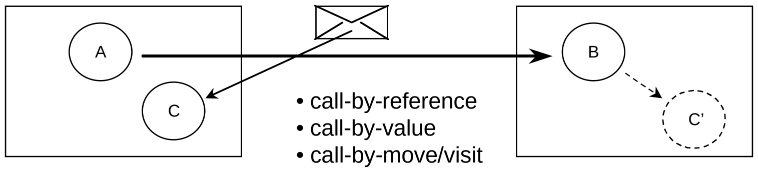


Code/objets/agents mobiles

- Code mobile
  - rapprocher (code) traitement des données
  - ex : SQL

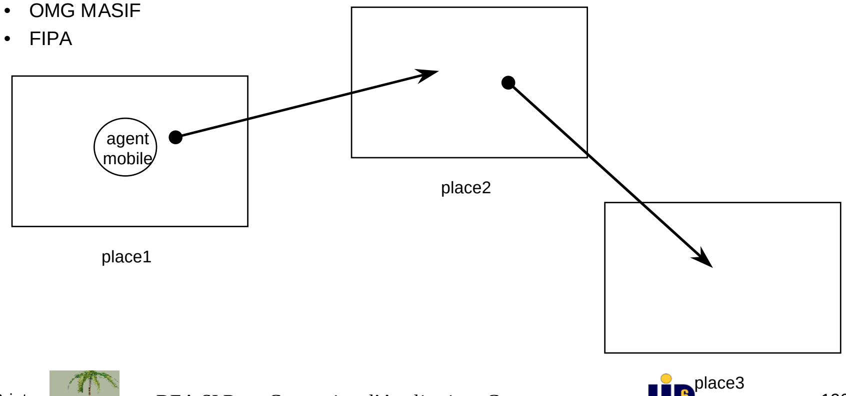


- Objet mobile
  - PostScript (code + données constantes)
  - Emerald [Black et al. IEEE TSE 87]



Agents mobiles

- A la différence du code ou de l'objet mobile, c'est l'agent mobile qui a l'initiative de son déplacement
- Langages :
  - Telescript (initiateur)
  - (Java-based) Odissey, Aglets, Voyager, Grasshopper, D'Agents (ex-AgentTcl), etc.
  - Standardisation :
    - OMG MASIF
    - FIPA

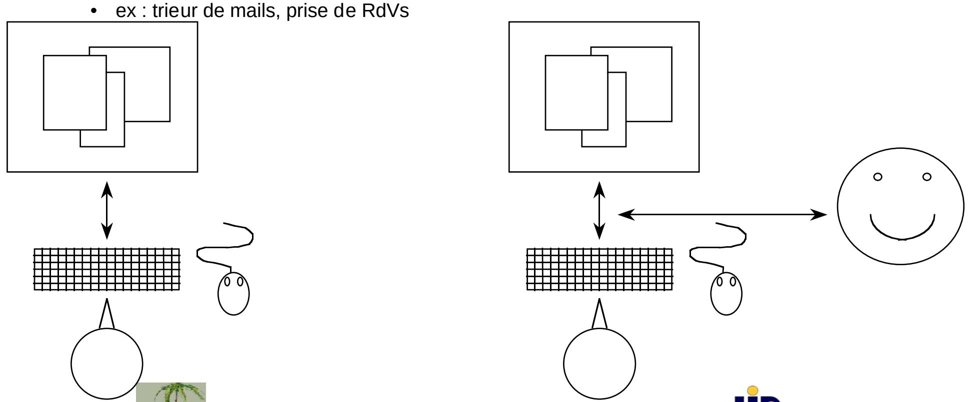


Agents mobiles (2)

- Avantages des (mis en avant par) les agents mobiles
  - Réduction du trafic (traitement local -> données échangées réduites)
    - agents mobiles vs RPC
  - Robustesse
    - Déconnexion du client mobile (informatique nomade : pause, tunnel, ombre…)
  - Confidentialité (traitement local)
    - (mais problèmes de sécurité)
  - Evolution logicielle
    - Off-line
      - Diffusion (versions) de logiciels (download)
    - On-line
      - Réseaux actifs
        - Données et Méta-données de contrôle (capsules)
- « Find the killer application ! »
  - Une nouvelle technique (parmi les) de programmation répartie
  - Combinaison (avec les autres : RPC, réplication, etc.) et non pas remplacement
- Recherches actuelles
  - Sécurité
  - Hétérogénéité
  - Collaboration (les agents mobiles actuels restent encore trop souvent solitaires)

Agents assistants

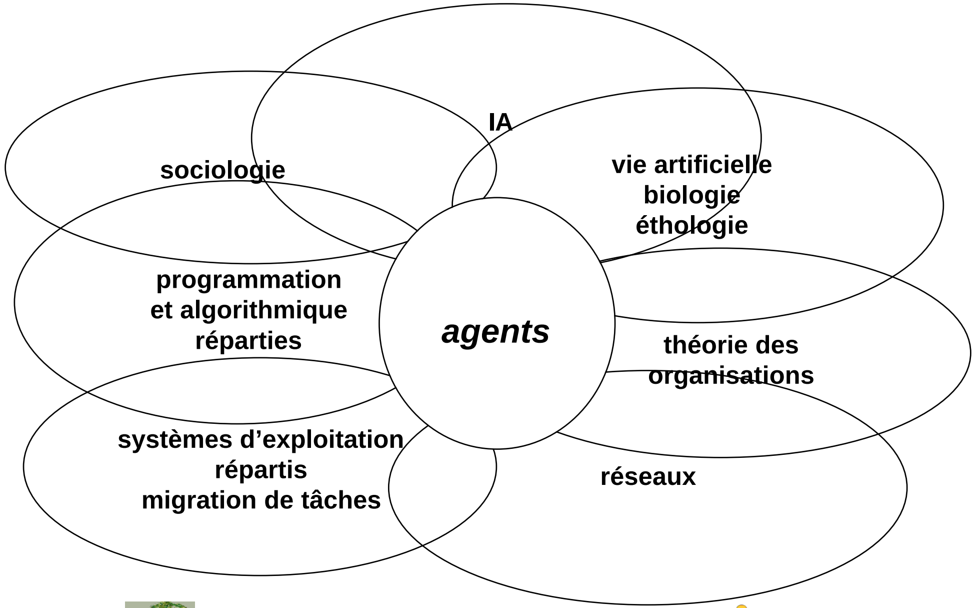
- Limitations des interfaces homme-machine classiques
  - à manipulation directe / explicite
  - rigidité, complexité, ne s'améliore pas à l'usage
- Agents assistants
  - adaptation au profil de l'utilisateur, automatisation de certaines tâches, rappel d'informations utiles, <u>initiative</u>
  - ex : trieur de mails, prise de RdVs



Agents assistants (2)

- Ex : Bargain Finder, Letizia, Firefly (MIT AI Lab)...
- « If you have somebody who knows you well and shares much of your information, that person can act on your behalf very effectively. If your secretary falls ill, it would make no difference if the temping agency could send you Albert Einstein. This issue is not about IQ. It is shared knowledge and the practice of using it in your best interests. » [Negroponte, Being Digital, 1995]
- Complémentarité (humain - agent)
  - Utilisateur : « lent » en calcul ; agent : « rapide »
  - Utilisateur : langage naturel et vision ; agent pas encore…
  - « Show what an agent what to do » vs « Tell an agent what to do »
  - Critique : agents of alienation [Lanier, 1995]
- Vers des agents assistants et coopératifs

agents



Différents niveaux d'agents (cf Les Gasser)

- ermites
  - représenter un humain
  - données+procédures (objet)+contrôle+ressources(processus) (acteur)
    - réactivité, autonomie
  - action persistante
    - pro-activité, mission
  - capacités entrées/sorties et communication
  - * mobilité
  - * apprentissage
- agents sociaux
  - langage de communication entre agents (KQML, ACL, XML...)
  - échange de données
  - tâches
  - modèle (représentations) des autres
- multi-agent
  - action collective
  - division du travail (spécialisation)
  - coordination/intégration (gestion des dépendances et de l'incertain)

agents cognitifs vs agents réactifs

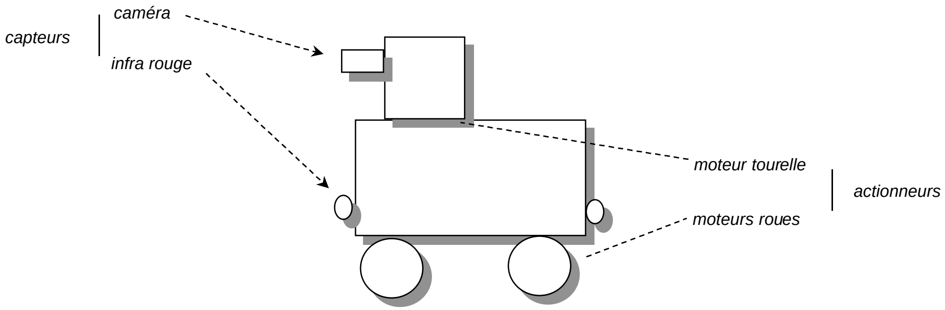
- agents cognitifs
  - représentation explicite
    - soi
      - connaissances (beliefs)
      - buts (intentions)
      - tâches
      - plans
      - engagements
    - environnement
    - autres agents
      - compétences
      - intentions
  - architectures complexes, souvent modèle logique (ex : BDI, Agent0)
  - organisation explicite
    - allocation et dépendances tâches
    - partage des ressources
    - protocoles de coordination/négociation
  - communication explicite, point à point, élaborée (ex : KQML)
  - petit/moyen nombre d'agents
  - top down, systématique
  - certaines validations formelles possibles

agents réactifs vs agents cognitifs

- agents réactifs
  - pas de représentation explicite
  - architectures simples
    - stimulus -> réponse
  - organisation implicite/induite
    - auto-organisation, ex : colonie de fourmis
  - communication via l'environnement
    - ex : perception/actions sur l'environnement, phéromones de fourmis
  - grand ou très grand nombre d'agents
    - redondance
  - robustesse
  - bottom up
  - validation expérimentale

Achitectures d'agents

- Architecture d'un agent = le "cœur" de l'agent, ce qui décide quoi faire
- Ex : architecture de contrôle d'un robot mobile autonome
  - problème clé : sélection de l'action (quoi faire ensuite ?)



- Propriétés/caractéristiques recherchées :
  - comportement à la fois délibératif et réactif
  - perception incertaine de l'environnement
  - robustesse (résistance aux pannes et aux dangers)
  - flexibilité de conception (boucle conception/évaluation)

Cela sera revu plus loin, à la lumière des architectures logicielles et des composants

Validation : la « grande » question

- Validations formelles
  - comportement individuel et collectif
    - modèles logiques, ex :
      - BDI [Georgeff et Rao] [Jennings et Wooldridge]...)
      - intentions jointes [Cohen]
  - coordination
    - réseaux de Petri, ex : [ElFallah]
  - négociation
    - théorie des jeux [Rosenschein...]
  - Mais en général contraint les modèles (certaines hypothèses de staticité, etc.)
- Validations semi-formelles
  - tests, couvertures de tests, invariants...
- Validations expérimentales
  - protocoles expérimentaux
  - reproductibilité des comportements et résultats observés
  - analyses de sensibilité (ex : aux conditions initiales)
  - attention aux influences des conditions d'exécution
    - ex : algorithme de séquencement, générateurs de nombres pseudo-aléatoires

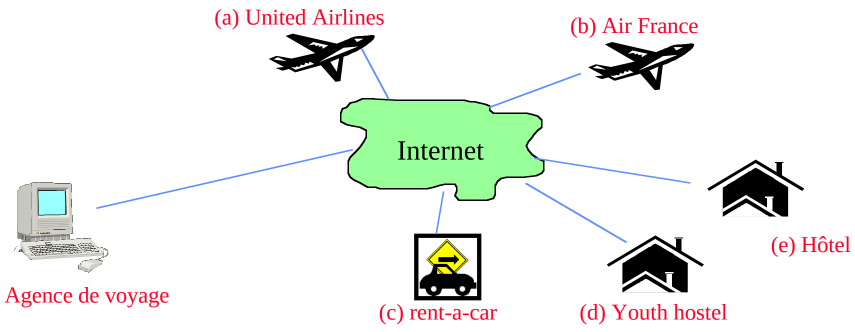
Agent, dans l'œil de l'observateur ??

- bilame d'un chauffe-eau
- test de Turing
- est-ce qu'un objet/processus (distribué ?) pourrait faire la même chose ??
- rationalité
- intentionnalité
  - comportement individuel
  - comportement collectif
- Canon de Morgan (1894) - psychologie comparative - éthologie
  - « En aucun cas, nous ne pouvons interpréter une action comme la conséquence d'un exercice ou d'une faculté psychique plus haute, si elle peut être interprétée comme l'aboutissement d'une faculté qui est située plus bas dans l'échelle psychologique »
  - -> behaviorism (explication causale) vs intentionnel (explication fonctionnelle)
- mesures quantitatives « objectives » ?
  - ex : ajout d'un agent -> pas de dégradation des performances (éventuellement amélioration) [Ferber 95]

Décomposition parmi les agents

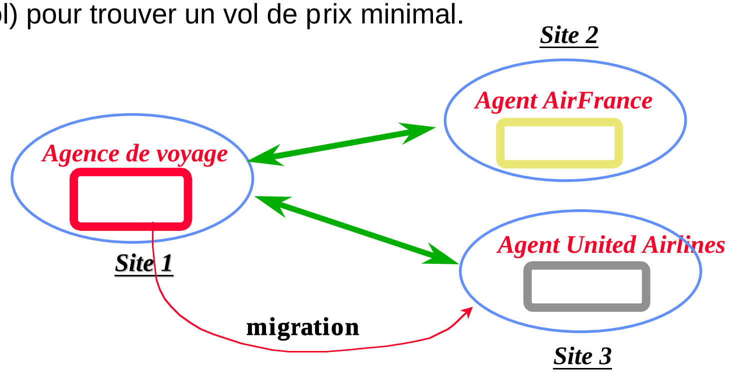
- décomposition des tâches, plans, sous-buts
- assignation aux agents
  - division du travail (spécialisation) vs totipotence
  - organisation, rôles
  - réseaux d'accointances
    - représentations des capacités des autres agents
  - appel d'offre
    - Contract Net protocol [Smith IEEE Transac. Computers 80]
  - market-based algorithms
    - mise aux enchères (protocoles : à la bougie, anglaise, hollandaise...)
  - formation de coalitions
    - (composition d'agents pour résoudre des tâches non faisables individuellement)

Ex : Scénario d'agence de voyage électronique [FIPA et projet CARISMA - thèse M.-J. Yoo, octobre 1999]



Exemple de protocole de coopération entre agents : choix du meilleur billet d'avion

- Deux agents serveurs de voyage, un agent agence de voyage
- Coopérer suivant un protocole d'appel d'offre (Contract net protocol) pour trouver un vol de prix minimal.



- Mobilité : l'agent se déplace vers le site du serveur choisi pour continuer la conversation (et optimiser les communications)

Organisations

- théorie des organisations - 3 points de vue [Scott 81] :
  - organisations rationnelles
    - collectivités à finalités spécifiques
    - objectifs, rôles, relations (dépendances...), règles
  - organisations naturelles (végétatives)
    - objectif en lui-même : survie (perpétuer l'organisation)
    - stabilité, adaptativité
  - systèmes ouverts
    - inter-relations/dépendances avec d'autres organisations, environnement(s)...
    - échanges, coalitions
- organisations d'agents (artificiels)
  - notion de rôle :
    - ex : client, producteur, médiateur ; attaquant, défenseur, gardien de but...
  - spécialisation des agents (simplicité vs flexibilité)
  - redondance des agents (efficacité vs robustesse)
  - relations
    - dépendances, hiérarchie, subordination, délégation
  - protocoles d'interaction/coordination
  - gestion des ressources partagées

Organisations (2)

- agents cognitifs
  - organisations explicites
- agents réactifs
  - organisations semi-implicites
    - façonnement de l'environnement, ex : fourmilière
    - « auto-organisation », ex : stigmergie des colonies de fourmis

- Exemple : extraction de minerai par des robots [Ferber 95]
- spécialisation ou pas des agents
  - totipotents (un agent sait jouer tous les rôles = sait tout faire)
  - rôles prédéfinis : robots détecteur, foreur, transporteur
- organisations du travail :
  - équipes (partenaires affectés statiquement)
    - ex : 1détecteur, 3 foreurs, 2 transporteurs
  - appel d'offre (partenaires affectés dynamiquement)
  - « émergentiste »
  - évolutives
    - feedback environnement, apprentissage, algorithmes génétiques...

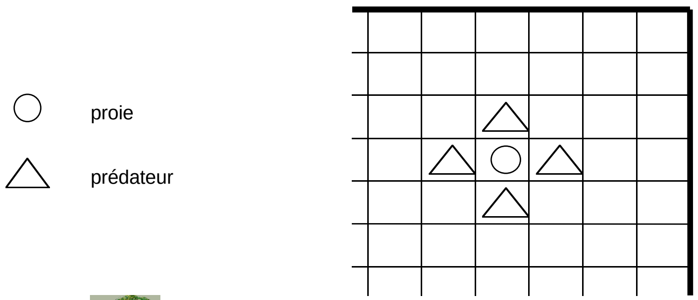
Coordination

- Motivations :
  - capacités individuelles insuffisantes (ex : charges trop lourdes à transporter)
  - cohérence (réguler les conflits sémantiques : buts contradictoires, accès aux ressources...)
  - efficacité (parallélisation de l'exécution des tâches)
  - robustesse, traitement de l'incertain
  - recomposition des résultats - solutions partielles

- Techniques
  - planification centralisée, semi-centralisée (synchronisation de plans individuels), distribuée, ex : Partial Global Plans [Durfee et Lesser IJCAI'87]
  - synchronisation d'accès aux ressources
    - algorithmique répartie
  - règles sociales
  - spécialisation (spatiale, objectifs…)
  - négociation
    - numérique, symbolique (agrégation, argumentation), démocratique (vote, arbitrage)
  - utilitarisme (théorie des jeux)
  - sans communication explicite
    - (environnement, reconnaissance d'intentions, de plans...)

Exemple des proies-prédateurs

- sur un environnement quadrillé, 4 prédateurs tentent d'encercler une proie
  - problème de coordination des mouvements des prédateurs
  - qualités : simplicité, généricité, efficacité, robustesse, propriétés formelles...
- approche cognitive
  - échange de plans (déplacements prévus), coordination
- approche réactive
  - attirance forte vers les proies, répulsion (faible) entre prédateurs



Communication

- environnement
  - perception, action (ex : consommation ressources)
  - traces (ex : phéromones)
- symbolique (messages)
  - medium (réseau, voix, vision...)
  - participants :
    - individuel - point à point
    - partagé - multicast
    - global - broadcast
    - publish/subscribe (événements)
    - par le contenu, Tuple-space, ex : Linda [Gelerntner 88]
- actes de langage - « dire c'est faire » [Searle 79]
  - composante locutoire
    - message, encodage
  - composante illocutoire
    - réalisation de l'acte de langage
    - performatifs : affirmer, questionner, annoncer, répondre...
  - composante perlocutoire
    - effets sur croyances des autres

Communication (2)

- Langages et protocoles de communication
- interoperabilité d'agents (CORBA des agents)
- KQML [Finin et Labrou 94] message
  - contenu
  - langage (d'expression du contenu)
    - ex : Java, Smalltalk, KIF, XML
  - ontologie
    - hiérarchie de concepts pour un domaine donné (ex : commerce e-, automobile...)
  - performatif (intention de la communication, lié à un type d 'interaction)
    - ex : ask, deny, register, recruit, request...

Note : beaucoup de choses implicites dans le monde objet deviennent explicites ici

- FIPA ACL (Agent Communication Language)
  - comme KQML, avec en plus :
  - sémantique formelle
  - protocole explicité
    - ex : FIPA-Contract-Net, FIPA-Iterated-Contract-Net

Limites (1/2) [Jennings 1999]

- No magic !
  - Un système développé avec des agents aurait probablement pu être développé avec des technologies plus conventionnelles
  - L'approche agent peut simplifier la conception pour certaines classes de problèmes
  - Mais elle ne rend pas l'impossible possible !
- Les agents sont des logiciels (presque comme les autres)
  - Principalement expérimental
  - Pas encore de techniques (é)prouvées
  - Ne pas oublier les aspects génie logiciel (analyse de besoins, spécification, conception, vérification, tests...)
  - Ne pas oublier les aspects concurrence/répartition
    - Problèmes (synchronisation…)
    - Mais également avantages (souvent encore peu exploités)
  - Réutiliser les technologies conventionnelles
    - Objets, CORBA, bases de données, noyaux de systèmes experts...
  - Utiliser les architectures agent existantes
    - Sinon vous passerez la majeure partie du temps dans la partie infrastructure et pas dans les spécificités des agents...

Limites (2/2)

- Trouver la bonne granularité
  - Equilibre à trouver entre : « un nombre est un agent » et « un seul agent dans le système »
  - *dans le monde objet : programmer une seule classe avec 1000 variables...*
  - Complexité vs modularité
- Importance de la structure (organisations, protocoles, connaissances…)
  - Il ne suffit pas de « jeter » ensemble des agents pour que cela fonctionne !
- Besoins en méthodologies
  - Cassiopée [Collinot & Drogoul 96]
  - Aalaadin/AGR [Ferber & Gutknecht 97]
  - Gaia [Jennings 99]
- Modélisation
  - Tentatives actuelles d'extension d'UML vers les agents (ex : AUML)
  - *Attention !* : UML est un ensemble de notations standardisée, et n'est *pas* une méthodologie

Vers des Méthodologies (analyse et conception) adaptées

- Find the agents !
  - Trop souvent, les agents sont (ou plutôt SEMBLENT) déjà donnés avant même l'étape d'<u>analyse</u>
    - ex : robots footballeurs
  - Mais, cela n'est pas toujours le cas
  - De plus, une identification (des agents) trop directe/intuitive ne sera pas forcément bénéfique dans la suite, car l'identification des agents :
    - quels concepts seront réifiés en agents
    - et lesquels ne seront pas !
    - quelle granularité...

  ...dépend beaucoup de l'objectif de la modélisation, des propriétés attendues...
- Cassiopée [Collinot et Drogoul 1996]
  - Objectif : Faire de la notion d'<u>organisation</u> l'objet véritable de l'analyse, qui peut être manipulée par le concepteur lors de la phase de conception, et/ou par les agents lors de l'exécution
  - Identifier les <u>dépendances</u> fonctionnelles entre les <u>rôles</u> (regroupement de comportements, mis en œuvre par des agents) qui sont inhérentes à l'accomplissement collectif de la tâche considérée.
  - Organisation : <u>gestion</u> (décentralisée et <u>dynamique</u>) <u>des dépendances (entre rôles)</u>

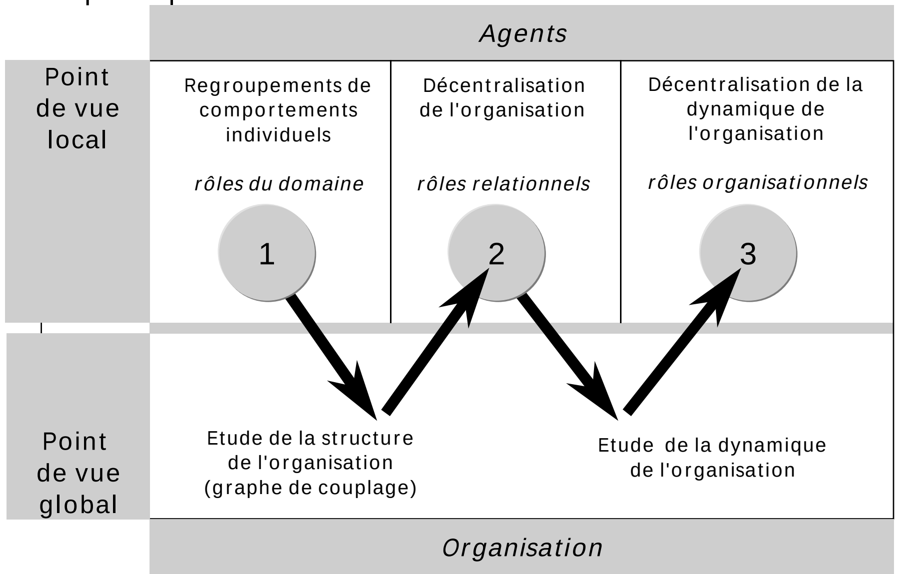
Cassiopée 1/2

- Un agent est composé d'un ensemble de rôles (3 différents niveaux)

	Rôles	*Typologie*	*Comportements*	*Signes échangés*
Agent	*dépendants du domaine*	*dépendant de l'application*	*dépendant de l'application*	–
	relationnels	agent influent	produit les signes d'influence en fonction du rôle du domaine	signes d'influence
		agent influencé	interprète les signes d'influence pour contrôler les rôles du domaine	
	organisationnels	initiateur	comportement de formation de groupe	signes d'engagement
			comportement de dissolution de groupe	
		participant	comportement d'engagement	signes de dissolution

Cassiopée 2/2

- Exemple de parcours : vers une méthode



Agents et objets (concurrents, distribués)

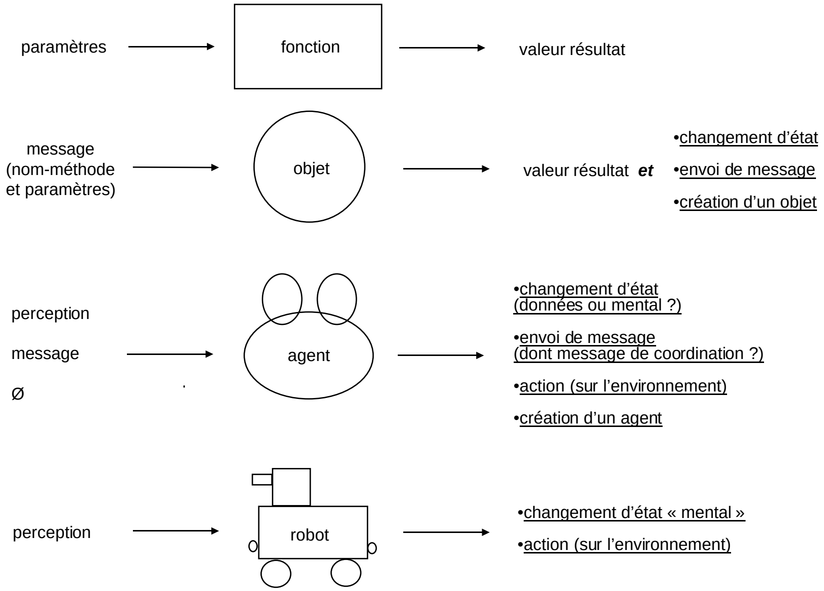
- OK, donc les agents semblent avoir des caractéristiques différentes ou supplémentaires des objets
  - au niveau des entités (pro-actives vs réactives, déclaratives vs procédurales...)
  - au niveau des organisations (adaptatives vs statiques et déterministes...)
- Regardons cela de plus près...

Différences entre Objets et Agents (1ère passe)

- au niveau de l'entité
  - agent non purement procédural
    - connaissances
      - ex : états mentaux, plans, règles d'inférence des agents cognitifs
  - pro-activité
    - pas uniquement purement réactif
- au niveau d'un ensemble d'agents
  - différents modes de communication
    - via l'environnement, ex : colonies de fourmis
    - messages typés, ex : KQML (inform, request, reply...)
  - coordination
    - interactions arbitrairement complexes, pas juste client/serveur
- au niveau de la conception (vs implantation)
  - organisation
    - structuration forte/explicite, souvent dynamique, conditionnant les interactions, la division du travail, les accès aux ressources partagées... : les rôles et leur coordination
  - une conception sous forme d'agents peut ensuite être réalisée sous forme d'objets ou d'acteurs, le niveau agent n'apparaissant plus explicitement dans l'implantation

Différences entre Objets et Agents (2ème passe)

- fonction (pure)
- objet
- agent
- robot

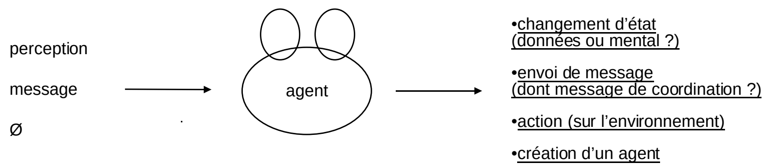


Bilan

- Le domaine des agents (agents logiciels, systèmes multi-agents...) est de fait encore relativement récent. Mais il aborde maintenant une nouvelle phase, des méthodes, des plates-formes de niveau pré-industriels sont maintenant proposées
- Quelque soit le type d'agent que nous envisagions, comment les construire ?
  - en ne réinventant pas la « roue » à chaque système
  - avec méthode et outils

Construire des agents

- Aspect essentiel du problème de la « sélection de l'action »
- Le calcul de cette sélection est a priori plus complexe que dans le cas des objets :
  - pas seulement procédural (ex : délibération)
  - nombreuses entrées (perception environnement, communication, coordination...)
  - « pro-activité » (et non plus juste « réactivité »), donc besoin d'arbitrage
  - mémoire complexe (ex : apprentissage)



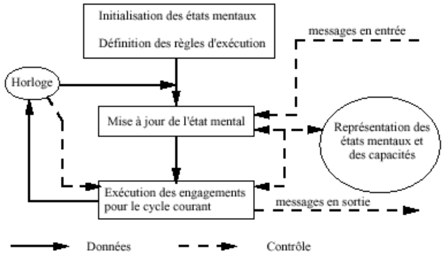
- On appelle communément <u>architecture</u> d'un agent la structure logicielle qui réalise cette sélection
- savoir si on inclut dans l'architecture ou pas les modules d'actions, ex : de communication n'est pas essentiel ici.

Construction des agents

- Comment programmer cette architecture ?
  - dans un langage spécifique
    - » ex : Agent0, April
    - » avantages :
      - (censé être) spécialisé
      - de plus haut niveau
    - » inconvénients :
      - incompatibilité avec les standards (Java, etc.)
      - un seul langage est-il de toute manière adapté ?
        - ex : langages de communication (ACLs)
  - dans un langage généraliste
    - » Java, Smalltalk, C++, Lisp...
    - » et c'est donc l 'architecture qui concrétise la structure
  - Note : on peut utiliser des langages spécifiques pour les différents modules
    - » ex : KQML/ACL pour la communication
    - » ex : AgentTalk, SCD pour la coordination

Agent languages

- April [McCabe et Clark 95]
  – basé sur Prolog concurrent (Parlog)
  – utilisé par Fujitsu (McCabe)
  – assez bas niveau, manque de structure
  – langage d'acteur mais avec des restes d'habits Prolog :)
- Agent0 [Shoham 93]
  – basé sur la notion d'états mentaux (croyances et engagements)
  – unification du cycle de raisonnement et de traitement des messages



  – implémentation en Lisp
  – (do <temps> <action>)
  – (inform <temps> <autre-agent> <fait>)
  – (commit <condition-message> <condition-état-mental> <action1> ... <actionN>)

Architectures

- Nous appelons architecture d'un agent, la structure logicielle (ou matérielle) qui, à partir d'un certain ensemble d'entrées, produit un ensemble d'actions sur l'environnement ou sur les autres agents. Sa description est constituée des composants (correspondant aux fonctions) de l'agent et des interactions entre ceux-ci (flux de contrôle) [Boissier 2001]

- Allons voir du côté des architectures logicielles (et des composants), domaines explorés indépendamment des agents
- Les motivations sont différentes : concevoir des programmes à grande échelle (« programming in the large ») et pouvoir raisonner sur l'assemblage (connexion, compatibilité, propriétés) de composants logiciels
- Mais les principes sont proches et ces travaux éclairent :
  – les organisations d'agents (mais couplage encore trop fort par rapport aux agents)
  – et également surtout les architectures d'agents (au niveau d'un agent : « programming in the small »)

Architectures logicielles et organisations (d'agents)

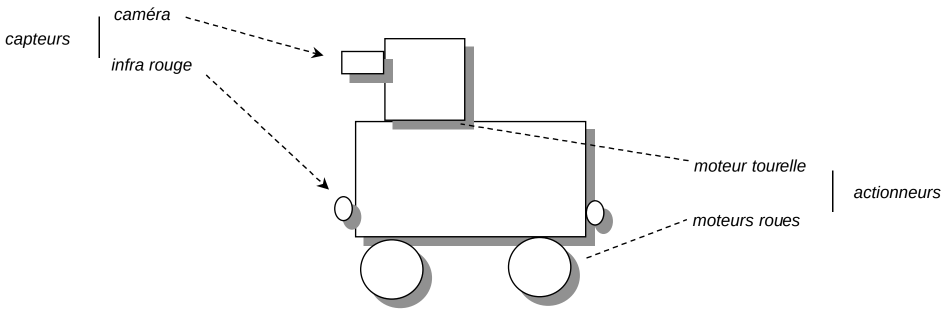
- Théorie (générale) des organisations - 3 points de vue [Scott 81] :
  – ***organisations rationnelles***
    - collectivités à finalités spécifiques
    - objectifs, ***rôles, relations (dépendances...), règles***
  – organisations naturelles (végétatives)
    - objectif en lui-même : survie (perpétuer l'organisation)
    - stabilité, adaptativité
  – systèmes ouverts
    - ***inter-relations/dépendances avec d'autres organisations, environnement(s)...***
    - échanges, coalitions

Architectures logicielles et organisations (d'agents) (2)

- Architectures logicielles
  – explicites
  – rationnelles
  – couplage explicite
    - au niveau des données (interfaces, typage)
    - et des modes d'interactions (connecteurs)
- Organisations d'agents (cognitifs)
  – explicites
  – rationnelles
  – couplage sémantique
  – réifiées
  – vers des organisations évolutives par elles-mêmes
- Organisations d'agents réactifs
  - bottom up émergentes (ex : société de fourmis)
  – conformantes top-down (cf. livre Alain Cardon, Conscience artificielle et systèmes adaptatifs, Eyrolles, 1999)

Achitectures d’agents - styles architecturaux (architectures logicielles)

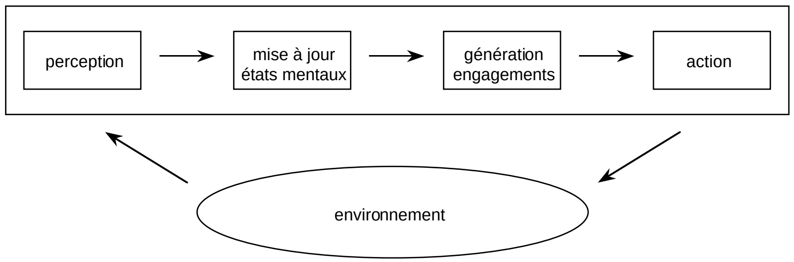
- Là, architecture = organisation individuelle / un agent (vision récursive)
- Exemple d ’application [Shaw et Garlan 96] :
  - (architecture de contrôle d’un) robot mobile autonome



- Propriétés/caractéristiques recherchées :
  - comportement à la fois délibératif et réactif
  - perception incertaine de l’environnement
  - robustesse (résistance aux pannes et aux dangers)
  - flexibilité de conception (boucle conception/évaluation)

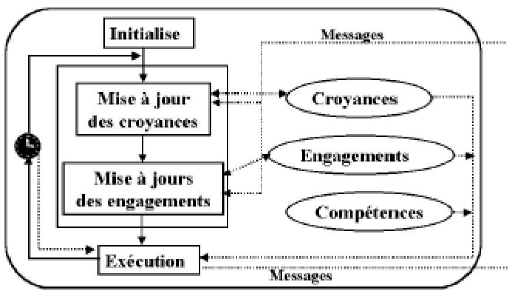
Architectures modulaires horizontales

- (une seule couche)
- cycle de calcul



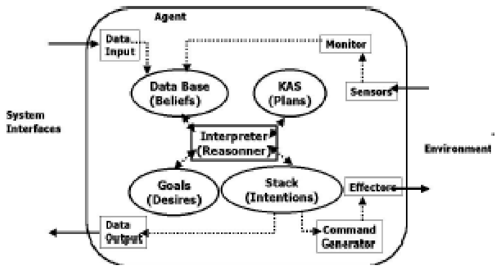
Architectures modulaires horizontales (2)

- **souvent basée sur notion d’états mentaux, engagements, intentions...**



AOP (Agent Oriented Programming) [Shoham 93]

plutôt dirigée par les états mentaux (data-driven)



PRS (Procedural Reasoning System) [Georgeff 87]

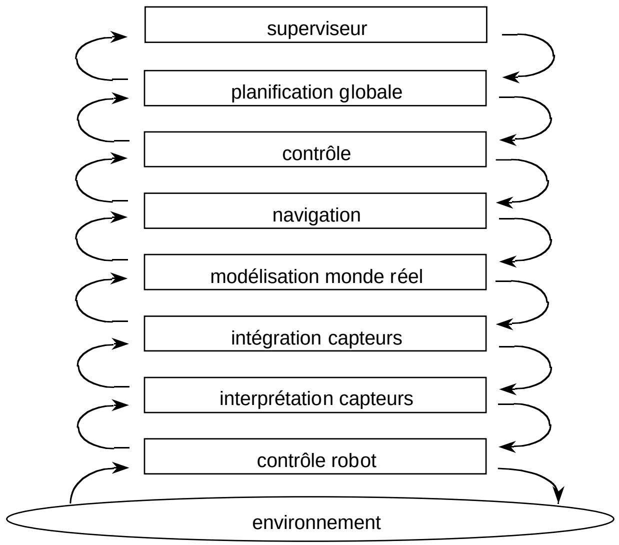
plutôt dirigée par les buts (goal-driven)

figures d’après [Boissier 2001]

Etats mentaux

- Etats mentaux
  - ex. d’architectures :
    - Agent0 [Shoham AI 93]
    - BDI [Rao et Georgeff 91]
  - formalisme logique (logique modale)
  - croyances
  - buts
    - comportements ou états désirés
  - plans
    - conditions de déclenchement
      - portant sur les croyances (data-driven) ou les buts (goal-driven)
    - actions ou sous-buts
  - intentions
    - intention = but persistant avec engagement d’accomplissement [Ferber@EcolIA’01]
    - intention = plan instancié (actif ou suspendu - en attente conditions)
    - intention = choix + engagement [Cohen et Levesque AI 90]
  - intentions jointes [Cohen et Levesque 95]
  - TeamWork [Tambe 99]

Architectures en couches (architectures verticales)



Architectures en couches (architectures verticales) (2)

- InteRRap [Müller 94]
- 3 couches activées en //
  - comportement - croyances sur état environnement
  - planification locale - croyances sur soi-même
  - planification coopérative - croyances sur et engagements avec les autres

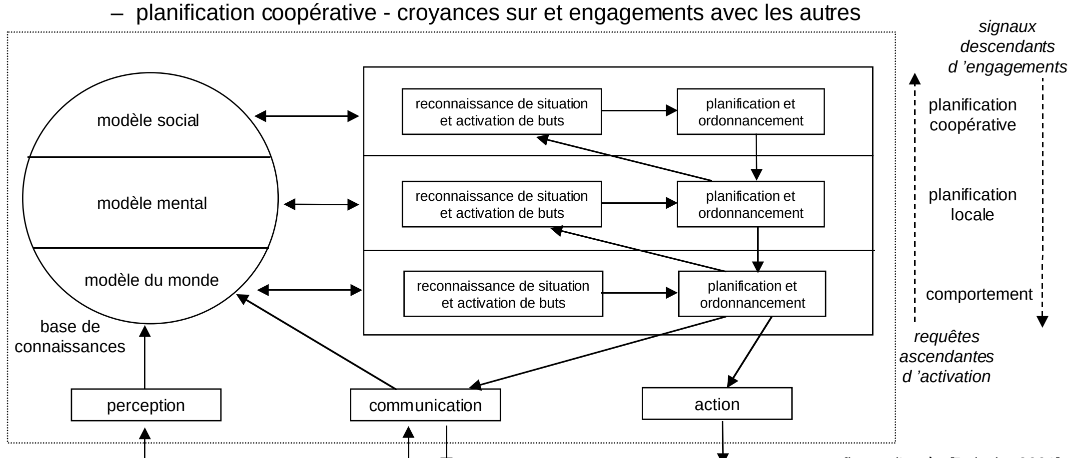


figure d'après [Boissier 2001]

Architectures en couches (architectures verticales) (3)

- TouringMachine [Ferguson 92]
- 3 couches activées en //
  - réaction
  - planification
  - modélisation (des entités y compris l'agent)
- contrôleur central - filtre perceptions et commandes (actions)
  - règles de contrôle (de censure et de suppression)

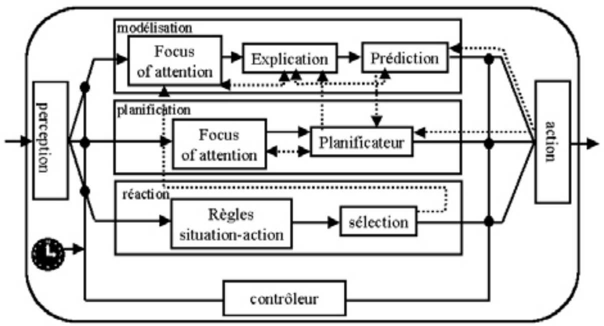
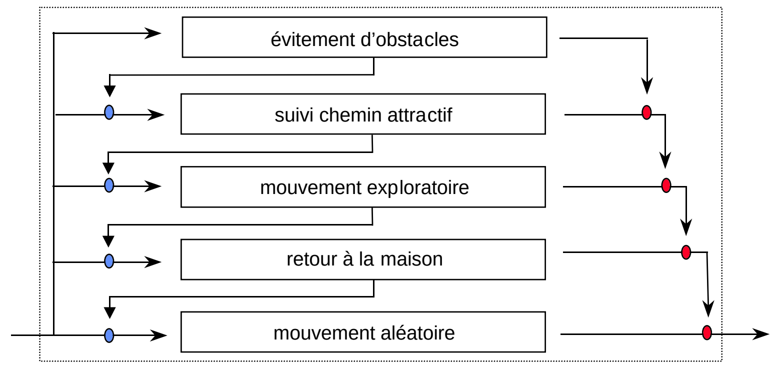


figure d'après [Boissier 2001]

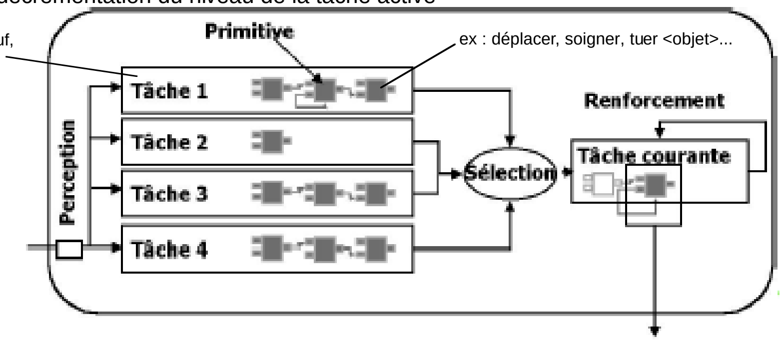
Architectures réactives en couches (verticales)

- « subsumption architecture » [Brooks 86]
- composants activés en parallèle
- compétition mais aussi hiérarchie
- priorités et inhibitions :
  - supplanter entrée composant inférieur
  - inhiber sortie composant inférieur



Architectures réactives en couches (2)

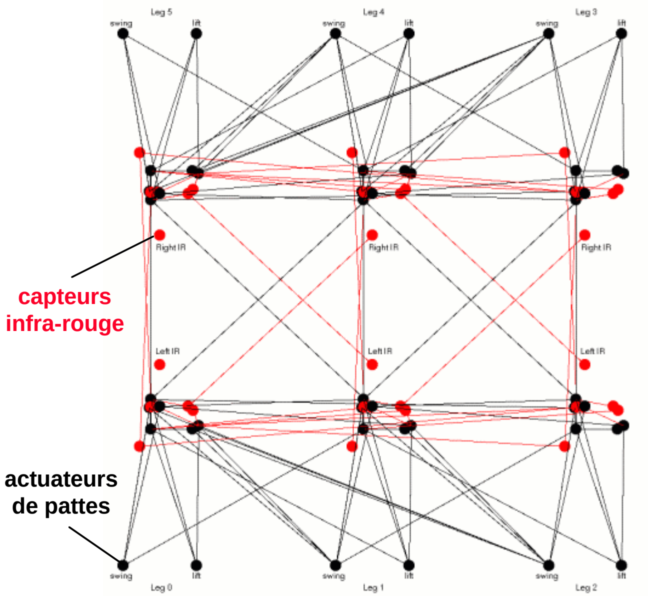
- MANTA [Drogoul 93]
- tâches indépendantes
  - poids (importance au niveau de l'agent)
  - niveau d'activation (calculé à partir du poids et de l'intensité des stimuli)
- sélection (par compétition) parmi les tâches
  - une (seule) tâche
    - nouvelle version (pour robots), primitives réflexes (ex : évitement d'obstacles) activables en //
  - niveau d'activation le plus élevé
  - décrémentation du niveau de la tâche active



Architectures d'agents

- hiérarchiques (cognitives/réactives)
  - ex : DIMA [Guessoum 96], InterRap [Müller 96]...
- componentielles
  - ex : Maleva [Lhuillier 98] [Meurisse 2000]
  - SCD [Yoo 98]
- composition d'actions
  - ex : Bene theory [Steels 94]
- connexionistes
- évolutionistes
  - algorithmes génétiques, morphogenèse

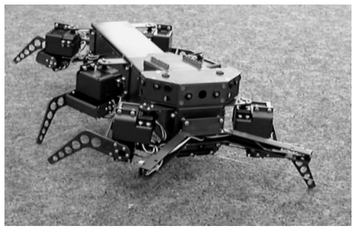
Evolution de l'architecture de contrôle d'un robot marcheur [Meyer et al. 98]



Evolution du programme de développement (instructions: DIVIDE, GROW, DRAW...)

Résultat :

Réseau noir : marche
Réseau rouge : évitement d'obstacle

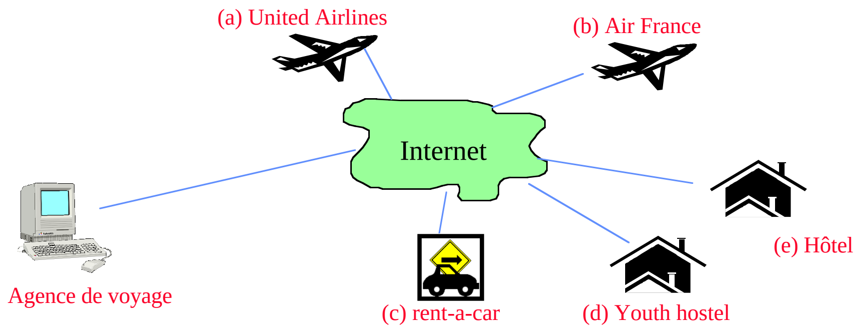


Plates-formes

- Architectures d'agents
- Bbiliothèques de
  - comportements
  - protocoles de coordination
- Outils
  - ex : noyaux de systèmes experts : JESS, JRules...
- Environnements de déploiement et d'exécution
- Environnements de visualisation et d'analyse des résultats
- Standard FIPA
- Plates-formes industrielles
  - Jack
  - AgentBuilder
  - Zeus, ...
- Plates-formes académiques
  - DIMA
  - MadKit
  - MASK, ...

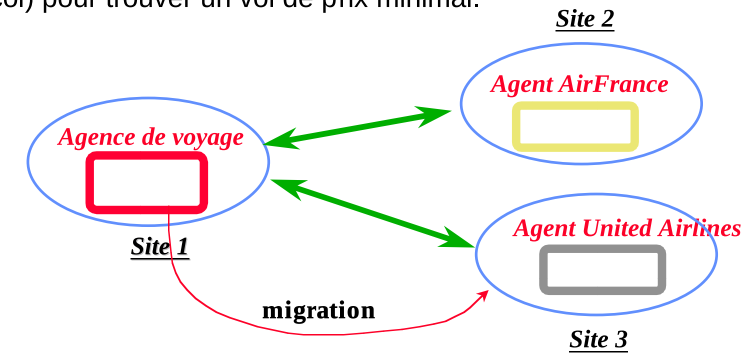
Ex : Framework et composants pour des agents mobiles coopératifs pour le commerce électronique [Yoo 1999]

Scénario de l'agence de voyage électronique (FIPA)



Exemple de protocole de coopération entre agents : choix du meilleur billet d'avion

- Deux agents serveurs de voyage, un agent agence de voyage
- Coopérer suivant un protocole d'appel d'offre (Contract net protocol) pour trouver un vol de prix minimal.

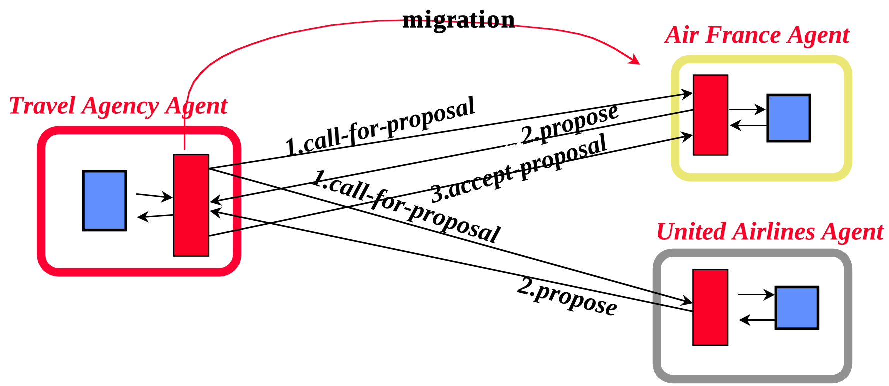


- Mobilité : l'agent se déplace vers le site du serveur choisi pour continuer la conversation (et optimiser les communications)

Coordination (Contract Net Protocol)

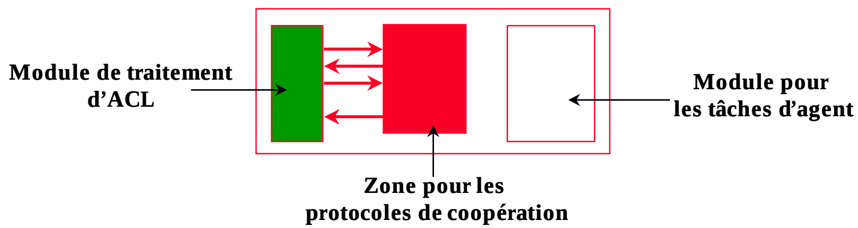
Composants (Java) pour les tâches spécifiques à l'agent

Composants SCD pour le protocole d'appel d'offre traduits en réseaux de Petri -> certaines validations



Framework d'agents

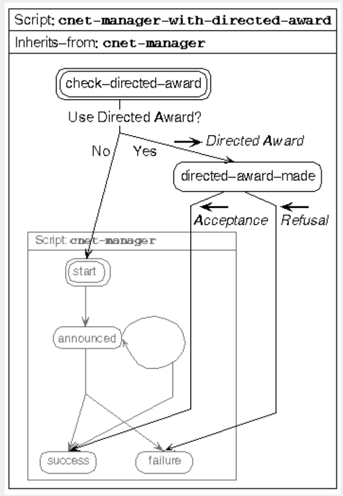
- Caractéristiques :
  - Dissocier le traitement du langage ACL du traitement du protocole de coopération
  - Le composant de traitement de l'ACL est proposé par le framework
  - Zone prédéfinie pour les protocoles de coopération
  - Connexions pré-établies entre le composant ACL et la zone pré-définie pour les protocoles de coopération



Facilite la modélisation

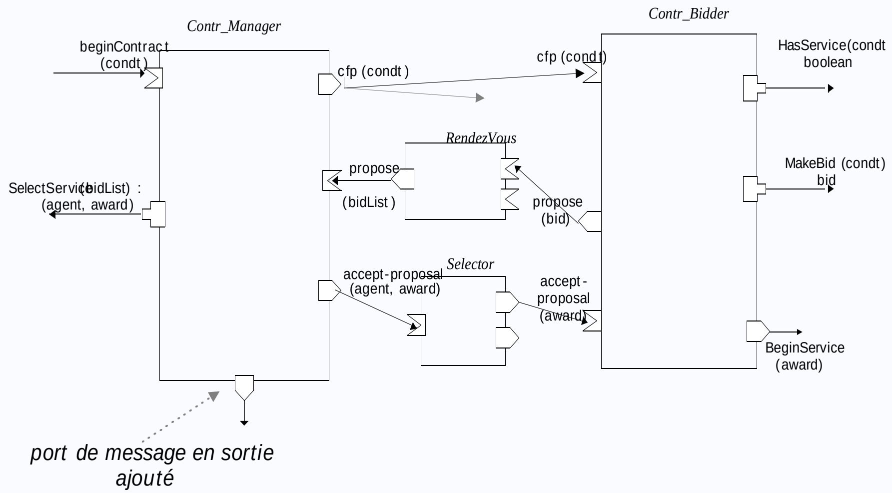
Réutilisabilité des composants (quelques résultats)

- Conception incrémentale de protocoles de coopération - à partir du Contract Net
  - par héritage
    - » ex : réalisation du protocole d'appel d'offre avec délai de temps : timeout Contract Net
  - par composition
    - » ex : extension en un FIPA-Iterated Contract Net
  - Expérimentations de réutilisation analogues avec AgentTalk (langage de coopération) par héritage [Kuwabara et al. 95]

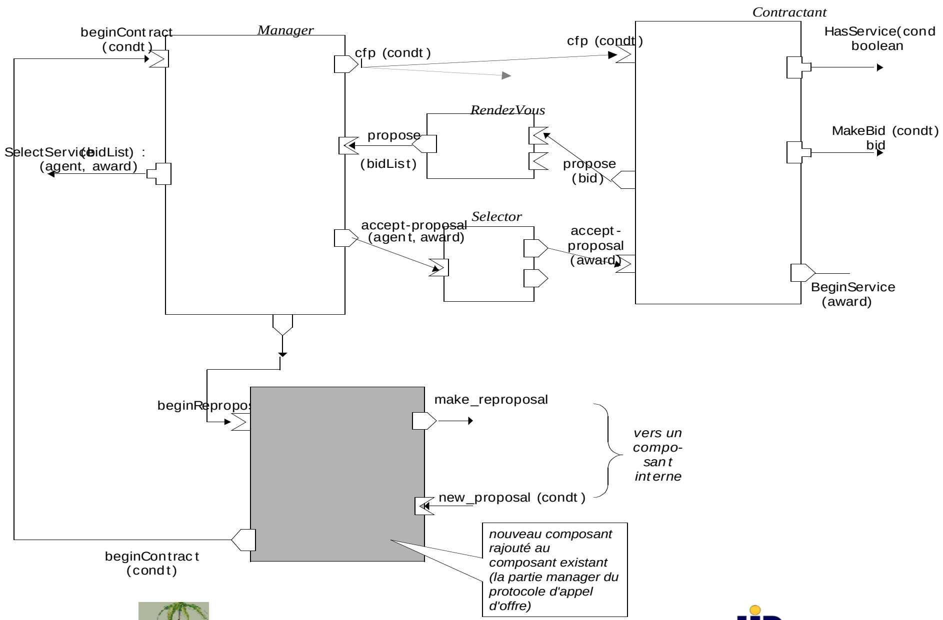


Raffinement de composants (par composition)

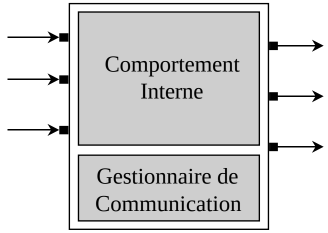
Iterated Contract Net Protocol

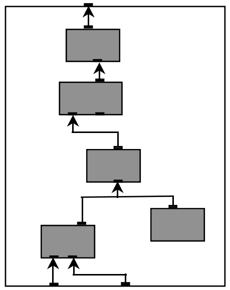


Raffinement de composants (par composition) (2)



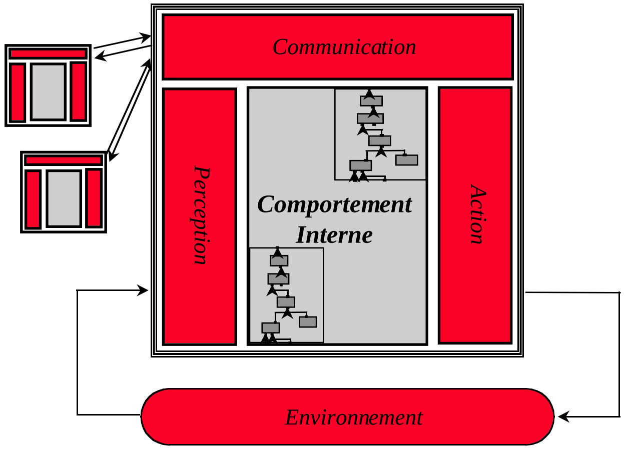
Composants d'agents Maleva [Lhuillier 98] [Meurisse 2000]





- Un Composant est défini par :
  - un Comportement Interne
  - des Bornes de Communication
- **Pas de référence explicite entre composants**
  - permet la modification du graphe des connexions indépendamment des composants.
  - entités *potentiellement* réutilisables car définition indépendante de l'environnement logiciel.

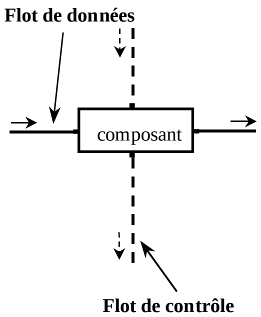
Maleva (2)



- Idée : modéliser le ***comportement interne*** des agents avec des composants

Maleva (3)

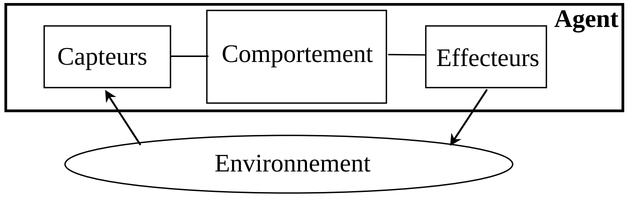
Principes :



- **Séparation explicite des flots de contrôle et de données**
  - Permet une plus grande généricité via l'expression de différents contextes de contrôle pour des mêmes composants
  - contrôle de haut niveau du séquencement (primordial pour limiter les biais de simulations)
- **2 types de bornes :**
  - ***Bornes de données***
    - Modification des ***variables d'instance*** du composant
  - ***Bornes de contrôle***
    - Un ***comportement encapsulé*** n'est enclenché que lors d'une activation via une borne de contrôle associée.

Exemples de Conception

Agents situés dans un écosystème simulé

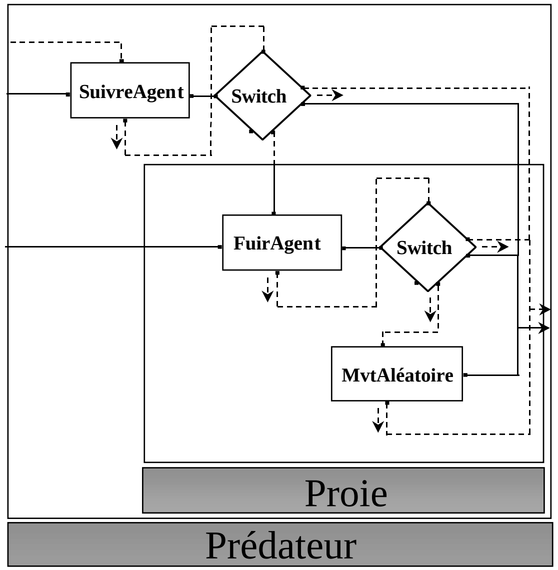


Exemple des Proies / Prédateurs

Les proies
- •Fuient les prédateurs
- •Mouvement aléatoire

Les prédateurs
- •Poursuivent les proies
- •Fuient les prédateurs
- •Mouvement aléatoire

Proies et prédateurs



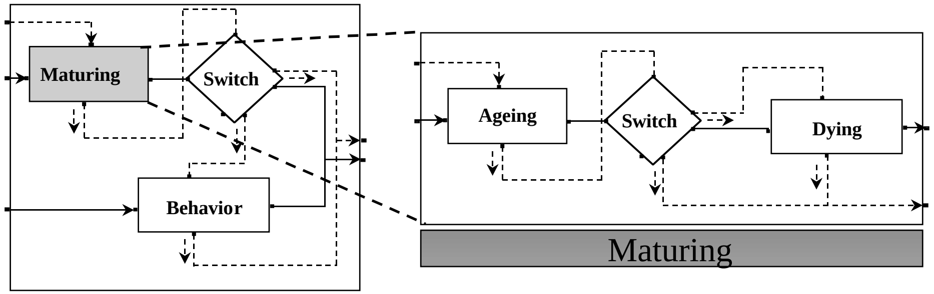
Exemples de Conception
Approche Descendante - *Les fourmis* [Guillemet et al. 98]

- Reformulation de MANTA [Drogoul 93] : simulation d'une colonie de fourmis
- Mise en évidence des notions de réutilisabilité et de dynamicité architecturale du modèle
- Différents types d'agents : reine, ouvrière, oeuf, larve
- Aspect <u>dynamique</u> des agents (et de leurs comportements) : passage de l'oeuf à la larve, de la larve à l'ouvrière
- Comportement commun de tout être vivant : agents évoluent en fonction du temps, naissent, vieillissent et meurent

-> ***<u>pattern</u>*** !

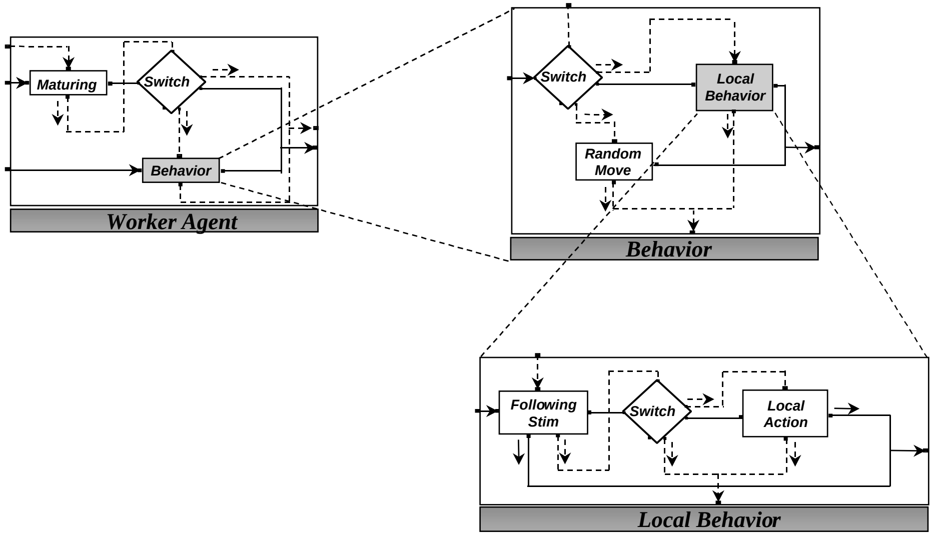
Maleva - les fourmis *(2)*

Comportement commun de tout être vivant : Le composant ***Maturing***
Agents évoluent en fonction du temps, naissent, vieillissent et meurent



Maleva - les fourmis *(3)*

Décomposition d'une ouvrière

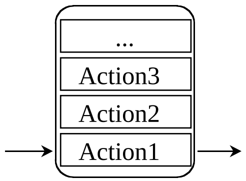


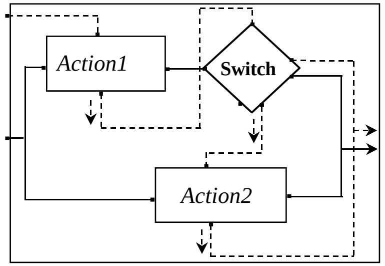
Maleva - Design Patterns

- Complexité potentielle des schémas de connexion de composants
- <u>Idée</u> : guider le concepteur d'architecture
  - Par contrainte : **typage de connexions**
  - Par aide à la conception : **design patterns**
- Recherche de *Design Patterns* à partir de schémas de composition récurrents

Maleva - Design Patterns (2)

- décomposition du flot de contrôle des langages impératifs : *if...then...else*
- architecture de Subsomption de Brooks [Brooks91]

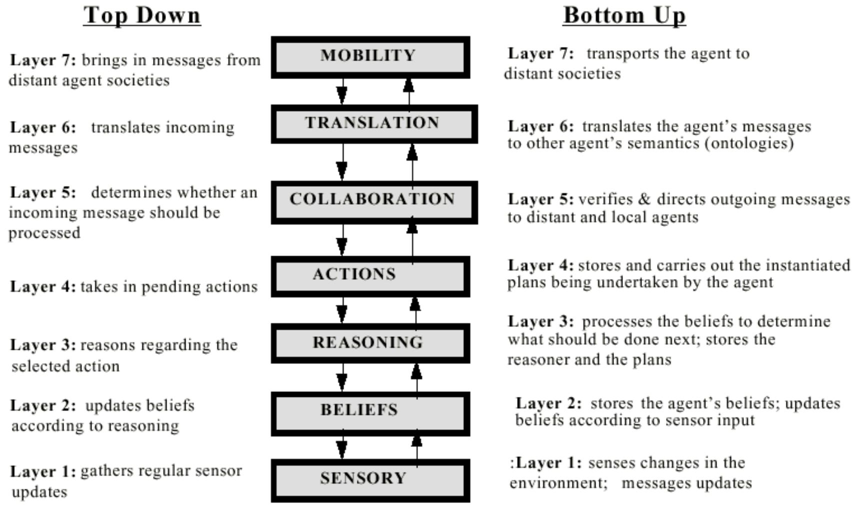




- capacité à vieillir (Composant ***Maturing***) dans des simulations d'écosystèmes

Autres Design patterns pour agents

- Marques [Sylvain Sauvage@EcolIA'01]
- Layered agent pattern [Kendall et al. 95]



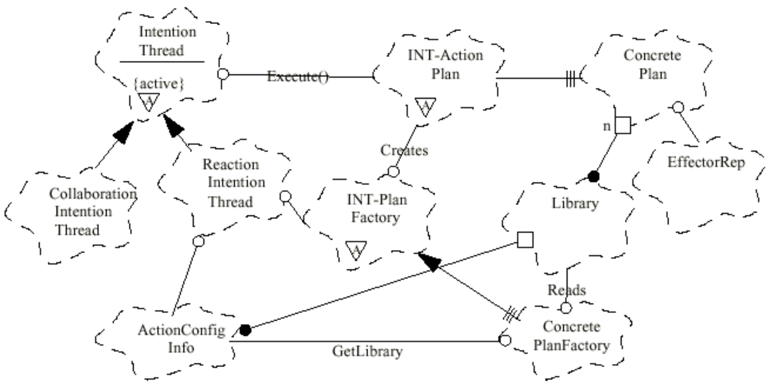
Plan as command [Kendall et al. 95]

Problem

How can a plan be encapsulated as an object ?

Forces

- Each Intention has a plan to execute. They have a wide range and are known only at run time.
- A plan specifies primitive actions, executed directly by the effectors or the Collaboration layer interface.
- There is a need to define a structure for plans that provides high level operations based on primitive ones.



Clone pattern [Kendall et al. 95]

Problem

How can an agent relocate itself and become resident in distant societies ?

Forces

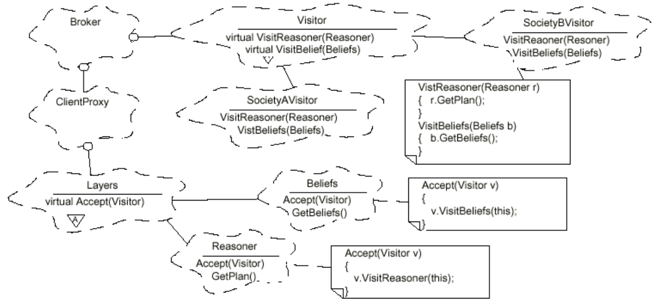
- An agent must be able to bring its capabilities, facilities, and state with it to a new society.
- The agent must be able to travel to a remote location and interact, negotiate, and exchange information in the new society.

Solution

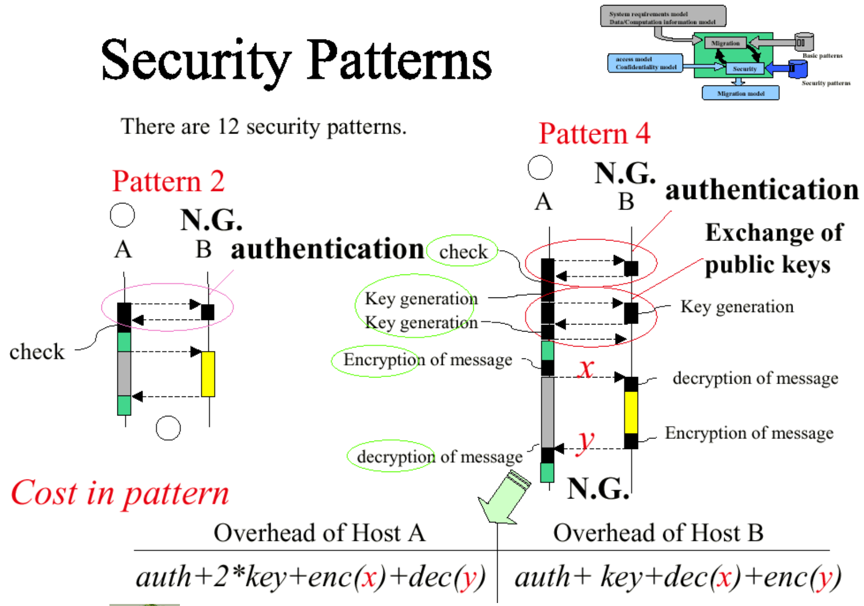
Make a copy or clone of the original agent, and place the new agent in the distant society. The clone must have all of the capabilities and facilities of the original agent, along with any state information.

Known Uses

The original use of agent self replication was cooperating mobile WAVE agents [3]. More recent approaches that utilize cloning include IBM Aglets [13], and the Agent Transfer Protocol (ATP) [14]. An aglet is a Java object that can move from one host on the Internet to another. When the aglet moves, it takes along its program code as well as its state (data). Bradshaw [5] refers to agent cloning as teleportation.



Security pattern for mobile agents [Honiden et al. 2000]



Plan : Conclusion

- Vers les agents
  - plus de sémantique
  - comportement et interactions arbitrairement complexes
  - couplage plus adaptatif entre entités
- Convergence nécessaire entre :
  - systèmes multi-agents
  - génie logiciel
  - algorithmique répartie

Ouvrages et pointeurs

- Les Systèmes Multi-Agents, Jacques Ferber, Interéditions, 1995
- *Software Agents, édité par Jeff Bradshaw, AAI-Press - MIT-Press, 1997*
- *Multi-Agent Systems, édité par Gerhard Weiss, MIT-Press, 1999*
- Principes et Architecture des Systèmes Multi-Agents, édité par Jean-Pierre Briot et Yves Demazeau, Collection IC2, Hermès, 2001

- *Revue Autonomous Agents and Multi-Agent Systems, Kluwer*

- www.multiagent.com
- www.fipa.org
- www.agentlink.org